CMU-CS-79-129

A Formal System for Specifying and Verifying Program Performance

Mary Shaw
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

21 June 1979

Abstract

Formal techniques for specifying performance properties of programs (e.g., execution time) and for verifying the correctness of these specifications are developed. These techniques are extensions of well-known predicate transformer techniques for specifying and verifying purely functional properties of programs.

This research was supported by the National Science Foundation under Grant MCS77-03883.

Table of Contents

1. Introduction

In order for a designer to completely describe a program or for a programmer to use it without undue concern for implementaton details, the program's specifications must address a wide variety of properties: what it computes, the resources it needs, its accuracy and reliability, and so on. Formal specifications are less ambiguous and more amenable to rigorous analysis than are informal specifications; when formal specifications can be written, they are often preferred for this reason. Formal techniques exist for specifying purely functional properties (i.e., what a program computes), but most other properties now admit of only informal or prose specification.

In this paper we address the problem of *formally specifying the performance properties of* programs. We measure performance in terms of resources consumed. Execution time and primary memory are the most obvious such resources, but secondary storage and file utilization, paging and I/O channel loads, and exclusive use of various devices could also be considered. We concentrate here on verifying statements about a program's *execution time*; Section 6 indicates the extension of this work to other resources.

Informal methods for estimating the time requirements of algorithms or programs are well known. They tend to focus on counting specific operations or entire paths in the program's flow graph [Aho74, Borodin75, Knuth73(sec 1.2.10)].[1] They provide estimates of a program's time requirements by arguing that the time required by a program is proportional to the operation or path count, and the estimates are often expressed as bounds. For many real programs, such rough estimates are not sufficiently precise -- the overhead of executing statements contributes significantly to the costs, and constant factors often *do* matter. Moreover, the informal methods are somewhat ad hoc: they are subject to human error and do not lend themselves directly to automatic techniques.

Our goal here is to present a precise, mechanizable system for

- specifying performance properties of programs (specifically execution time) as precisely as desired and
- formally verifying the consistency of those specifications with the associated programs.

Our technique is an extension of an existing technique for verifying purely functional properties. We extend the language semantics to include execution time, and we modify the

[1] This is the common pattern. There are some cases in which exact analyses are carried out by manually counting machine-level instructions [Knuth69(sec 4.6.3)].

proof rules (*predicate transformers*) so that they automatically manipulate assertions about execution time.

We begin with an example in which we add explicit counter variables to a program (Section 2). This serves as motivation for the extended predicate transformers (proof rules) in Section 3. Section 5 sketches several ways to control the precision of the estimates. The potential benefits and limitations of the technique are discussed in Section 6.

1.1. Examples

The techniques in this paper will be illustrated with two familiar programs for calculating x^n. The "slow" exponentiation program, which we call *SlowExp*, performs n repeated multiplications and returns the result in y:

```
y := 1;
for i := 1 to n do y := y * x
```

The "fast" exponentiation program, called *FastExp*, computes successive squares of the base and accumulates the appropriate ones into the result y:

```
y := 1; i := n; z := x;
while i > 0 do
   begin
   if i mod 2 ≠ 0 then y := y * z;
   z := z * z;
   i := i div 2
   end
```

The verifications of these programs are familiar exercises [Manna74, Manna77, Wulf80]. In both cases, the verification consists of showing that the precondition "$n>0$" leads to the postcondition "$y=x^n$". A loop invariant for *SlowExp* is

$$y=x^{i-1} \wedge n>0$$

while a loop invariant for *FastExp* is

$$x^n = y * z^i \wedge n \geq 0$$

The programming language of this discussion is closely related to Pascal. The proof rules are given as predicate transformers [Dijkstra76], rather than as axioms [Hoare72].

2. Measuring Execution Time with Programmed Counters

The traditional approach to determining program complexity calls for counting the number of times various operations or program paths are executed. The precision of the analysis can be controlled, in part, by the selection of which operations are to be counted. We will motivate the formal system to be developed in Section 3 by augmenting program *SlowExp* with a counter variable and some extra statements to tally the costs of operations as they occur. We will include accounting for such things as fetching, storing, and loop overhead as well as for the primary operation, multiplication. Instead of counting the occurrences of individual operations in separate variables, we maintain a single cost counter and increment it by a different amount for each kind of operation.

We introduce the extra variable π to accumulate the time estimate, and we let symbolic constants Δ_{op} represent the costs of operations. The costs of the operations in each statement are assessed by incrementing π in an adjacent statement. The augmented program, *AugSlowExp*, is[1]

```
y := 1;  π := π + Δ:=;                {1 assignment}
π := π + Δforst + Δget;               {evaluate upper bound and start loop}
for i := 1 to n do
   begin
   π := π + Δfor;                     {continue loop}
   y := y * z;
   π := π + Δ* + Δ:= + 2Δget          {2 fetches, 1 multiplication, 1 assignment}
   end
```

[1] We assume that fetching integer constants and proceeding from one statement to the next are free.

Note that we charge the cost of loop initialization (Δ_{forst}) just once, but we charge for the increment-test-and-repeat operations (Δ_{for}) on each iteration.

It is not necessary to execute this program in order to verify a statement about its execution cost -- an assertion about the cost (that is, about the final value of π) can be included in the postcondition, and standard verification techniques can be used to prove the assertion. The complete pre and postconditions for *AugSlowExp* are

$$\{\ n>0 \wedge \pi=0\ \}\ \text{AugSlowExp}\ \{\ y=x^n \wedge \pi = n(\Delta_{*}+\Delta_{:=}+2\Delta_{get}+\Delta_{for}) + \Delta_{:=}+\Delta_{forst}+\Delta_{get}\ \}$$

To verify these extended assertions we need to extend the loop invariant by adding a term involving π; the invariant becomes

$$y=x^i \wedge n>0 \wedge \pi = \sum_{j=1}^{i-1}(\Delta_{*}+\Delta_{:=}+2\Delta_{get}+\Delta_{for}) + \Delta_{:=} + \Delta_{forst} + \Delta_{get}$$

With this invariant, the verification of *AugSlowExp* is straightforward.

Notice that we can keep this analysis machine-independent by using the symbolic constants, or we can make it machine-specific by using numeric values for the Δ_{op}. Note also that the assertions about π need not always be of the form "π=expression"; we might assert, for example, that "$n(\Delta_{*}+\Delta_{:=}+2\Delta_{get}+\Delta_{for}) \le \pi \le 2n(\Delta_{*}+\Delta_{:=}+2\Delta_{get}+\Delta_{for})$".

Inserting accounting statements in a program in this fashion is mechanical but tedious. It is subject to human error, especially if it accounts for many distinct kinds of operations. We will take advantage of the mechanical character of the transformation and address the problem of human error by extending the proof rules of the programming language to perform the accounting automatically.

3. A Formal System for Verifying Execution Costs

The informal approach of estimating costs by adding counters to a program can be captured precisely in a form similar to ordinary predicate transformers (proof rules) by extending the ordinary rules for computing weakest preconditions. This is the intuition behind the formal system developed in this section.

The control flow in a program (and hence the execution cost) may depend on the values of program variables. Specifically, the path through a conditional statement or the number of executions of a loop body may in general depend on current values of variables. This

dependence makes it necessary to maintain state information about the program during the performance analysis. As a result we cannot simply "count operations"; we must express the costs in terms of the values of variables. We deal with this problem by capturing both the relevant state information and the cost term in the same logical predicate that carries assertions about the program's functional properties.

3.1. The Computational Model

Just as the ordinary proof rules for a language define the semantics of the language, the augmented rules presented here extend the semantic definition to include execution costs. In order to do this, we make some simplifying assumptions about the nature of the costs involved and we adapt the semantic definitions to express the (simplified) computational model.

Our model of execution costs is simplified in two ways. First, we assume that all of the execution cost of a program can be attributed to discrete, identifiable events; that is, we assume that all costs are triggered by constructs that appear in the program text. Second, we assume that the cost of an operation is independent of the context in which it is invoked; that is, the cost of an operation may depend on the program state but not on the syntax of the adjacent statement. These assumptions are inherent to any system that estimates costs by counting operations. Some of the consequences are discussed in section 6.

The formulation of execution cost semantics presented here is based on a Pascal-like semantic definition stated in terms of predicate transformers. Since predicate transformers deal only with program constructs that modify the program state, we add execution cost to the program state. When we do this, expressions are no longer free of effects on the state, so we must add proof rules for expressions.

3.2. Predicate Transformers for Execution Costs

We introduce a pseudo-variable, π, for each program. Intuitively, π corresponds to the program clock; it will be incremented by some suitable amount each time an operation is performed. The assertions of the program may involve π as well as the program variables, but the program text may not manipulate π directly. The predicate transformer for each program construct increments π in addition to performing the transformation appropriate to the construct. We call the result a *cost precondition* instead of a *weakest precondition*. In our performance verifications, the theorem to be proved about a program S will be of the form

$$\{P \wedge \pi=0\}\ S\ \{Q \wedge \pi=cost\}$$

where P and Q are predicates involving only program variables. The proof of such a theorem involves proving that

$$(P \wedge \pi=0) \supset cp(S, Q \wedge \pi=cost)$$

where cp(S,R) is the cost precondition of statement S and assertion R. The syntactic form of the assertion that separates the π term from the rest of the assertion is used here to simplify the exposition; it is not essential to the theory.

For example, consider a theorem about a program that consists of a single assignment statement:

$$\{t=3 \wedge \pi=0\} \quad x:=t \quad \{x=3 \wedge \pi=\Delta_{:=}+\Delta_{get}\}$$

This theorem states that if the program is executed when t has the value 3 and no cost has been assessed, then after the program terminates, x will have the value 3 and a cost of $\Delta_{:=}+\Delta_{get}$ will have been assessed.

The cost predicate rules often require substitutions in the predicates we use the notation "P_b^a" to denote the expression that results from substituting the expression a for every occurrence of the identifier b in formula P. Simultaneous substitution for multiple variables is permitted. The cost precondition rule for assignment statements is

$$\underline{cp}(x := E, Q) \equiv \underline{cp}(E, Q)_{x \quad \pi}^{E \quad \pi+\Delta_{:=}}$$

This is the usual rule for assignment, except for two things. First, the substitution of $\pi+\Delta_{:=}$ for π has the effect of tallying the cost of the assignment; second, the invocation of a cost predicate on the right-hand side accounts for the cost of evaluating the expression E. The cost precondition rule for fetching the value of a simple variable is

$$\underline{cp}(x, Q) \equiv Q_{\pi}^{\pi+\Delta_{get}}$$

This rule affects only the term involving π, not the remainder of the assertion. Applying the two cost predicate rules in turn, we give the proof of this program in full detail.

$(t=3 \wedge \pi=0) \supset \underline{cp}(x:=t,\ x=3 \wedge \pi = \Delta_{:=}+\Delta_{get})$ theorem to prove

$(t=3 \wedge \pi=0) \supset \underline{cp}(t,\ x=3 \wedge \pi = \Delta_{:=}+\Delta_{get})_{t\ \pi}^{x\ \pi+\Delta_{:=}}$ use assignment rule

$(t=3 \wedge \pi=0) \supset (x=3 \wedge \pi+\Delta_{get} = \Delta_{:=}+\Delta_{get})_{t\ \pi}^{x\ \pi+\Delta_{:=}}$ use fetch rule and substitute $\pi + \Delta_{get}$

$(t=3 \wedge \pi=0) \supset (t=3 \wedge \pi+\Delta_{:=}+\Delta_{get} = \Delta_{:=}+\Delta_{get})$ substitute x and $\pi + \Delta_{:=}$

which follows from simple arithmetic.

To see the cumulative effect of applying these rules to a program with several statements, consider a simple interchange program,

$$\{x=a \wedge y=b \wedge \pi=0\}\ \ t:=x;\ x:=y;\ y:=t\ \ \{y=a \wedge x=b \wedge \pi = 3\Delta_{:=}+2\Delta_{;}+3\Delta_{get}\}$$

We need the cost precondition rule for sequential statements,

$$\underline{cp}(S1;\ S2,\ Q) \equiv \underline{cp}(S1,\ \underline{cp}(S2,\ Q)_{\pi}^{\pi+\Delta_{;}})$$

as well as the rules for assignment and variable fetch given above. We compute the cost precondition of the program, then show that it follows from the given precondition. We abbreviate steps that repeat the proof above, and we perform arithmetic simplification on the equation involving cost constants whenever possible.

$\underline{cp}(t:=x;\ x:=y;\ y:=t,\ \ y=a \wedge x=b \wedge \pi = 3\Delta_{:=}+2\Delta_{;}+3\Delta_{get})$

$\equiv \underline{cp}(t:=x;\ x:=y,\ \underline{cp}(y:=t,\ \ y=a \wedge x=b \wedge \pi = 3\Delta_{:=}+2\Delta_{;}+3\Delta_{get})_{\pi}^{\pi+\Delta_{;}})$ use sequence rule

$\equiv \underline{cp}(t:=x;\ x:=y,\ \ t=a \wedge x=b \wedge \pi = 2\Delta_{:=}+\Delta_{;}+2\Delta_{get})$ do "y:=t" and subst for ";"

$\equiv \underline{cp}(t:=x,\ \underline{cp}(x:=y,\ \ t=a \wedge x=b \wedge \pi = 2\Delta_{:=}+\Delta_{;}+2\Delta_{get})_{\pi}^{\pi+\Delta_{;}})$ use sequence rule

$\equiv \underline{cp}(t:=x,\ \ t=a \wedge y=b \wedge \pi = \Delta_{:=}+\Delta_{get})$ do "x:=y" and subst for ";"

$\equiv x=a \wedge y=b \wedge \pi = 0$ do "t:=x"

We complete the proof by noting that the precondition given in the theorem about the program is identical to the computed cost precondition.

Note that the performance verification, like any verification based on weakest preconditions, propagates assertions from the end of the program toward the beginning. Further, the substitutions for π performed by the cost precondition rules have the effect of

incrementing π. If no simplifications are done, the result is to generate intermediate assertions with the form

π + (time from here to the end) = (total time)

However, if arithmetic simplifications are performed, the (equivalent) assertions will be more suggestive of

π = (time from beginning to here)

which may be intuitively more appealing. The assertions for the example of Section 2 are in the latter form.

3.3. Cost Precondition Definitions

In this section we state the cost precondition rules for a subset of Pascal.[1] The costs of operations are represented symbolically as Δ_{op}, and π is the pseudo-variable used for time accounting. The interpretation of other symbols is

S	statement	unop	unary operator
E	expression	binop	binary operator
B	boolean expression	f	function
V	vector	x	simple variable
H, I, Q	predicate	c	constant

The cost preconditions for statements are derived from the corresponding weakest-precondition rules by adding substitutions that increment π and continuing the computation through any embedded statements or expressions.

$$\underline{cp}(S1; S2, Q) \equiv \underline{cp}(S1, \underline{cp}(S2, Q)_{\pi}^{\pi+\Delta_;})$$

$$\underline{cp}(x := E, Q) \equiv \underline{cp}(E, Q)_{x\ \ \pi}^{E\ \ \pi+\Delta_{:=}}$$

---

[1] The omissions are the case statement, parameters to routines, the goto, and declarations. The case statement is a simple extension. Parameters, gotos, and declarations present more issues with respect to verification than with respect to cost; the functional semantics of these constructs should be extendible to execution time with relatively little difficulty. In Pascal, declaration cost can be absorbed into procedure-call overheads but this is not true for all block-structured languages. Cost predicates for declarations in general need to reflect time required for allocation and initialization.

$\underline{cp}(V[E1] := E2, Q) \equiv \underline{cp}(V[E1], \underline{cp}(E2, Q))^{E2}_{V[E1]}{}^{\pi+\Delta:=}_{\pi}$

$\underline{cp}(\underline{assert}\ R, Q) \equiv R \wedge Q$

$\underline{cp}(\underline{if}\ B\ \underline{then}\ S1\ \underline{else}\ S2, Q) \equiv (B \wedge \underline{cp}(S1, \underline{cp}(B, Q))^{\pi+\Delta ift}_{\pi}) \vee (\neg B \wedge \underline{cp}(S2, \underline{cp}(B, Q))^{\pi+\Delta ife}_{\pi})$

$\underline{cp}(\underline{while}\ B\ \underline{do}\ S, Q) \equiv \bigvee_1^{\infty} H_i$

where $H_0 = \underline{cp}(B, \neg B \wedge Q)^{\pi+\Delta whst}_{\pi}$

$H_i = B \wedge \underline{cp}(B, \underline{cp}(S, H_{i-1}))^{\pi+\Delta wh}_{\pi}$

$\underline{cp}(\underline{for}\ i := E1\ \underline{to}\ E2\ \underline{do}\ S, Q) \equiv \underline{cp}(E1, \underline{cp}(E2, I[E1..E1]^{\pi+\Delta forst}_{\pi}))$

provided $(I[E1..i-1] \wedge E1 \le i \le E2) \supset \underline{cp}(S, I[E1..i])^{\pi+\Delta for}_{\pi} \quad \wedge \quad I[E1..E2+1] \supset Q$

and $I[a..b]$ means that the invariant $I(i)$ holds for $a \le i < b$

$\underline{cp}(\underline{call}\ f, Q) \equiv (P_f \wedge (Q_f \supset Q))^{\pi+\Delta call}_{\pi}$

provided the specifications of f are P_f { body of f } Q_f

(P_f and Q_f are assumed to contain expressions involving π)

We must also provide cost precondition rules for expressions in order to account for time consumed in expression evaluation. We assume that costs of function calls are the same as for procedure calls. We also consider operators (binary and unary) and fetches (of constants, simple variables, and vector elements). Times for all the binary and unary operators are assumed to be constant; Section 6 discusses ways to relax this assumption.

$\underline{cp}(E1\ binop\ E2, Q) \equiv \underline{cp}(E1, \underline{cp}(E2, Q))^{\pi+\Delta binop}_{\pi}$

where "binop" may be +, -, *, /, <, >, ≤, ≥, =, etc.

$\underline{cp}(unop\ E, Q) \equiv \underline{cp}(E, Q)^{\pi+\Delta unop}_{\pi}$

where "unop" may be -, ¬

$$\underline{cp}(x, Q) \equiv Q_{\pi}^{\pi+\Delta_{get}}$$

$$\underline{cp}(c, Q) \equiv Q_{\pi}^{\pi+\Delta_{const}}$$

$$\underline{cp}(V[E], Q) \equiv \underline{cp}(E, Q)_{\pi}^{\pi+\Delta_{ss}}$$

In the remainder of this paper, we will assume $\Delta_{;} = \Delta_{const} = 0$ and that $\Delta_{ift} = \Delta_{ife} = \Delta_{if}$. We also use a simplified form of the while rule, in which an invariant is used as an approximation to the cost precondition.

The simplified while rule resembles the normal predicate transformer. It may be used when certain auxiliary quantities can be determined. Suppose that

I is a predicate not involving π or any Δ,

j is a new variable that counts executions of the loop body,

K(j) is the cost of the j^{th} loop body,

M is the number of times the loop is executed,

$\pi+K(j)=C \wedge I$ {if B then S} $\pi = C \wedge I$

Then the following formula may be used as the cost precondition for the while loop:

$$\underline{cp}(\underline{while}\ B\ \underline{do}\ S,Q) \equiv \underline{cp}(B, I \wedge \pi + \sum_{j=1}^{M} K(j)=C)_{\pi}^{\pi+\Delta_{whst}}$$

provided $(I \wedge \pi+\sum_{j=i}^{M} K(j)=C \wedge B) \supset \underline{cp}(S, \underline{cp}(B, I \wedge \pi+\sum_{j=i+1}^{M} K(j)=C)_{\pi}^{\pi+\Delta_{wh}})$

and $(\neg B \wedge I \wedge \pi=C) \supset Q$

Note that an exact analysis of the program requires an exact count, M, of the number of times the loop body is executed. Section 5 addresses some of the situations in which only a bound or an approximation is available.

4. Examples

We now prove performance results for the two examples introduced in section 1.1. We prove in particular that the cost of program *SlowExp* is

$$\Delta_{SlowExp} = n\,(\Delta_{:=}+\Delta_{*}+2\Delta_{get}+\Delta_{for}) + \Delta_{:=}+\Delta_{forst}+\Delta_{get}$$

and that the cost of program *FastExp* is

$$\begin{aligned}\Delta_{FastExp} = {} & 3\Delta_{:=}+\Delta_{whst}+\Delta_{>}+3\Delta_{get} \\ & + \sum_{j=1}^{\lceil \log n+1 \rceil} ((\lfloor n/2^{j-1} \rfloor \bmod 2)(\Delta_{:=}+\Delta_{*}+2\Delta_{get}) \\ & \quad +\Delta_{wh}+\Delta_{if}+\Delta_{\neq}+\Delta_{mod}+2\Delta_{:=}+\Delta_{*}+\Delta_{div}+\Delta_{>}+5\Delta_{get})\end{aligned}$$

These (somewhat formidable) expressions carry enough precision to obtain very accurate estimates of execution times for programs in a language that satisfies the assumptions of section 3.1. Often, however, cruder estimates are entirely acceptable. In section 5 we will consider ways to simplify these cost expressions with approximations and coarser descriptions.

4.1. Slow Exponentiation

We begin with program *SlowExp*; we retain the assertions about functionality as well as the assertions for performance, although the complete assertions are not necessary if only the performance result is desired. The program, annotated with preconditions, postconditions, and a loop invariant, is

$\{\ n>0 \wedge \pi=0\ \}$

y := 1;

for i := 1 to n do

$\quad \{\ y=x^{i-1} \wedge n>0 \wedge \pi+\sum_{i}^{n}(\Delta_{:=}+\Delta_{*}+2\Delta_{get}+\Delta_{for}) = \Delta_{SlowExp}\ \}$

$\quad$ y := y * x;

$\{\ y=x^{n} \wedge \pi=\Delta_{SlowExp}\ \}$

By convention, assertions about for loops hold at the beginning of the loop body, before the increment and test. A degenerate form of the invariant must hold before the first execution.

We begin by checking the loop invariant, first computing cp(loop body, $I[1..i])_{\pi}^{\pi+\Delta_{for}}$ and then checking the condition

$$((I[1..i-1] \wedge 1 \le i \le n) \supset cp(\text{loop body}, I[1..i])_{\pi}^{\pi+\Delta_{for}}) \wedge (I[1..n+1] \supset y=x^n \wedge \pi=\Delta_{SlowExp})$$

The cost precondition for the loop body is

$$cp(y:=y*x,\ y=x^{i-1} \wedge n>0 \wedge \pi+\sum_i^n(\Delta_{:=}+\Delta_{*}+2\Delta_{get}+\Delta_{for})=\Delta_{SlowExp})_{\pi}^{\pi+\Delta_{for}}$$
$$\equiv y=x^{i-2} \wedge n>0 \wedge \pi+\sum_{i-1}^n(\Delta_{:=}+\Delta_{*}+2\Delta_{get}+\Delta_{for})=\Delta_{SlowExp}$$
$$\equiv I[1....i-1]$$

In $I[1..n+1]$ the sum is empty, so the condition we must show to establish the invariant is

$$((I[1..i-1] \wedge 1 \le i \le n) \supset I[1..i-1]) \wedge ((y=x^n \wedge \pi=\Delta_{SlowExp}) \supset (y=x^n \wedge \pi=\Delta_{SlowExp}))$$

which clearly holds.

To complete the proof, we must show that the precondition of the program leads to the cost predicate of the loop; that is,

$$(n>0 \wedge \pi=0) \supset cp(y:=1, cp(1, cp(n, I[1..1]_{\pi}^{\pi+\Delta_{forst}})))$$

This reduces to

$$(n>0 \wedge \pi=0) \supset (1=x^0 \wedge n>0 \wedge \pi+n(\Delta_{:=}+\Delta_{*}+2\Delta_{get}+\Delta_{for})+\Delta_{:=}+\Delta_{forst}+\Delta_{get}=\Delta_{SlowExp})$$

or

$$n\,(\Delta_{:=}+\Delta_{*}+2\Delta_{get}+\Delta_{for}) + \Delta_{:=}+\Delta_{forst}+\Delta_{get} = \Delta_{SlowExp}$$

and completes the proof.

4.2. Fast Exponentiation

We turn now to the more complex program, *FastExp*. This time the assertions will include only terms that pertain to the program's performance.

$\{\ n>0 \wedge \pi=0\ \}$

$y := 1;\ i := n;\ z := x;$

<u>while</u> $i > 0$ <u>do</u>

<u>begin</u>

$\{\ n>0 \wedge \pi + \sum_{j=1}^{\lceil \log\, i+1 \rceil} (\mathrm{Bit}(i,j-1)(\Delta_{:=}+\Delta_{*}+2\Delta_{get}) + \Delta_{wh}+\Delta_{div}+\Delta_{*}+2\Delta_{:=}+\Delta_{if}+\Delta_{\neq}+\Delta_{mod}+5\Delta_{get}+\Delta_{>}) = \Delta_{FastExp}\ \}$

<u>if</u> i <u>mod</u> $2 \neq 0$ <u>then</u> $y := y * z;$

$z := z * z;$

$i := i$ <u>div</u> 2

<u>end</u>

$\{\ \pi=\Delta_{FastExp}\ \}$

Here, the function Bit(a,b) finds the b^{th} bit from the right end of a, so $\mathrm{Bit}(a,b) = \lfloor a/2^b \rfloor \bmod 2$. To simplify the expressions in our calculations, we let

$$\Delta_{inner} = \Delta_{wh}+\Delta_{div}+\Delta_{*}+2\Delta_{:=}+\Delta_{if}+\Delta_{\neq}+\Delta_{mod}+5\Delta_{get}+\Delta_{>}$$

denote the cost of the portion of the loop body that is executed on every iteration. We also denote the entire invariant as I.

We begin again by checking the loop invariant. The invariant is in a form suitable for the simplified <u>while</u> rules, the cost of each loop body is expressible, and the loop will be executed $\lceil \log\, n+1 \rceil$ times. We therefore choose the simplified form of the rule. We first compute $\underline{cp}(S, \underline{cp}(B, I))_{\pi}^{\pi+\Delta_{whst}}$, then we check the condition

$$((i>0 \wedge I) \supset \underline{cp}(S, \underline{cp}(i>0, I))_{\pi}^{\pi+\Delta_{wh}}) \wedge ((i\leq 0 \wedge I) \supset \pi = \Delta_{FastExp})$$

The cost precondition for the loop body includes the cost for reevaluating the condition for

each iteration. That cost is assessed first, then substitutions are performed for the two assignment statements and the expressions on the right-hand sides. Finally the if rule is applied and Δ_{wh} is added in. After a bit of arithmetic, the cost precondition for the loop body $\underline{cp}(S, \underline{cp}(i>0, 1))_{\pi}^{\pi+\Delta_{wh}}$ becomes

$$n>0 \ \wedge \ \pi + (i \bmod 2)(\Delta_{:=}+\Delta_{*}+2\Delta_{get})+\Delta_{inner}$$
$$+\sum_{j=1}^{\lceil \log((i \text{ div } 2)+1)\rceil}(Bit(i \text{ div } 2,j-1)[\Delta_{:=}+\Delta_{*}+2\Delta_{get}]+\Delta_{inner}) = \Delta_{FastExp}$$

Given that i>0, (i mod 2)=Bit(i,0), and Bit(i div 2,k)=Bit(i,k+1), this reduces to the invariant. The first term of the condition therefore holds. The second part of the condition follows from the observation that i=0 at the end of the final iteration so the sum is empty.

To complete the proof, we must show that the precondition of the program leads properly to the loop, or that

$$n>0 \wedge \pi=0 \ \supset \ \underline{cp}(y:=1;\ i:=n;\ z:=x,\ \underline{cp}(i>0,\ 1)_{\pi}^{\pi+\Delta_{whst}})$$

This expands to

$$n>0 \wedge \pi=0 \supset$$
$$n>0 \wedge \Delta_{FastExp} = [\pi + 3\Delta_{:=}+\Delta_{whst}+\Delta_{>}+3\Delta_{get}$$
$$+ \sum_{j=1}^{\lceil \log n+1\rceil}(Bit(n,j-1)(\Delta_{:=}+\Delta_{*}+2\Delta_{get}) + \Delta_{inner})]$$

and then to

$$\Delta_{FastExp} = 3\Delta_{:=}+\Delta_{whst}+\Delta_{>}+3\Delta_{get}$$
$$+ \sum_{j=1}^{\lceil \log n+1\rceil}(Bit(n,j-1)(\Delta_{:=}+\Delta_{*}+2\Delta_{get})$$
$$+\Delta_{wh}+\Delta_{div}+\Delta_{*}+2\Delta_{:=}+\Delta_{if}+\Delta_{\neq}+\Delta_{mod}+5\Delta_{get}+\Delta_{>})$$

which completes the proof.

5. Simplifications

The calculations specified by the rules above are precise, but they can become complex. In many cases, they provide more precision than the application requires. If the analysis is being carried out by automatic methods (e.g., in conjunction with a verification of functional properties), this may simply result in extra expense. Manual analysis, however, can be error-prone. In this section we indicate some ways cost analyses can be simplified and indicate a tradeoff between the amount of precision desired and the difficulty of the calculation.

Three kinds of simplifications for cost analyses are readily available. First, the description of the language can be simplified. Second, the cost estimates themselves can be made less precise by making looser assertions about the cost. Third, if assertions adhere to some simple syntactic assumptions, the processing of loops and routine calls can be simplified. We sketch these ideas here and leave the development for later papers. In several cases we will consider simplifications of the cost of *FastExp*. Recall that the exact cost was

$$\begin{aligned}\Delta_{FastExp} &= 3\Delta_{:=}+\Delta_{whst}+\Delta_{>}+3\Delta_{get} \\ &\quad + \sum_{j=1}^{\lceil \log n+1 \rceil} (\lfloor n/2^{j-1} \rfloor \bmod 2)(\Delta_{:=}+\Delta_{*}+2\Delta_{get}) \\ &\qquad +\Delta_{wh}+\Delta_{if}+\Delta_{\neq}+\Delta_{mod}+2\Delta_{:=}+\Delta_{*}+\Delta_{div}+\Delta_{>}+5\Delta_{get})\end{aligned}$$

5.1. Language Description

The cost precondition rules can be made specific to a particular implementation by determining values for the symbolic constants and performing the analyses with those values instead of with the symbolic constants. In a future paper we will take this approach and compare the costs predicted by cost precondition analyses with actual program timings.

Even symbolic computations can be made simpler by using a coarser set of symbolic constants. For example, we might define only four constants and make the following approximations:

Δ_{get}	is trivial and can be ignored	
Δ_1	replaces	Δ_{+}, Δ_{-}, $\Delta_{>}$, $\Delta_{=}$, $\Delta_{\neq}$, Δ_{whst}, Δ_{wh}, Δ_{if}
Δ_2	replaces	$\Delta_{:=}$, Δ_{forst}, Δ_{for}, Δ_{ss}, Δ_{val}, Δ_{var}, Δ_{*}, Δ_{r+}, Δ_{r-}, Δ_{r*}
Δ_3	replaces	Δ_{mod}, Δ_{div}, Δ_{call}

Under this simplification, the cost of the fast program *FastExp* becomes

$$\Delta_{FastExp} = 2\Delta_1+3\Delta_2+\sum_{j=1}^{\lceil \log n+1\rceil}(Bit(n,j-1)(2\Delta_2)+4\Delta_1+3\Delta_2+2\Delta_3)$$

5.2. Approximate Assertions

Simple analyses from concrete complexity often account only for a few major or indicative operations and assume that the execution time of the entire program can be predicted by the simpler counts. We can achieve the same effect by setting Δ_{op} to 0 for all except the major operations. For example, we can obtain a multiplication count for program *FastExp* in this fashion; it is

$$\Delta_{FastExp} = \sum_{j=1}^{\lceil \log n+1\rceil}(Bit(n,j-1)+1) = \lceil \log n+1\rceil + \text{number of 1-bits in } n$$

Another kind of simplification can be obtained by writing assertions that bound π without specifying an exact value. For example, it is easy to eliminate the Bit function from the *FastExp* example and show only that

$$\Delta_{FastExp} \le 3\Delta_{:=}+\Delta_{whst}+\Delta_{>}+3\Delta_{get} + \sum_{j=1}^{\lceil \log n+1\rceil}(\Delta_{wh}+\Delta_{if}+\Delta_{\neq}+\Delta_{mod}+3\Delta_{:=}+2\Delta_{*}+\Delta_{div}+\Delta_{>}+7\Delta_{get})$$

Also, if we had not known exactly how many times the loop in *FastExp* would be executed but we had been able to find a bound on the number of iterations, we could have expressed that partial knowledge as an inequality.

These simplifications can, of course, be combined. For example, we can describe the number of multiplications required by *FastExp* as

$$\log n \le \Delta_{FastExp} \le 2\lceil \log n+1\rceil$$

5.3. Special Forms of Assertions

These techniques for verifying performance properties make no restrictions on the way the performance estimate π, enters the assertions. In our examples, it has appeared in a separate term with the form $\pi + k1 = k2$, but the cost predicates apply equally well to arbitrary expressions involving π. Only the simplified *while* rule has required π to appear in an expression with a particular form, and that requirement was made in order to provide a convenient way to deal with a common special case.

We can establish a similar special case for the *for* statement. Again, let K(i) be the cost of the i^{th} loop body, and assume that the invariant has the form

$$I[E1..i] \wedge \pi + \sum_{j=i}^{E2} K(j) = C$$

Then

$$\underline{cp}(\underline{for}\ i:=E1\ \underline{to}\ E2\ \underline{do}\ S,Q) \equiv \underline{cp}(E1, \underline{cp}(E2, I[E1..E1] \wedge \pi + \sum_{j=E1}^{E2} K(j) + \Delta_{forst} = C))$$

$$\text{provided } (I[E1..i-1] \wedge E1 \le i \le E2) \supset I[E1..i] \wedge (I[E1..E2+1] \wedge \pi = C) \supset Q$$

This corresponds closely to the common intuition about the cost of a loop. When the cost of the body and the bounds are all constant, the cost expression simplifies further to

$$\underline{cp}(\underline{for}\ i:=C1\ \underline{to}\ C2\ \underline{do}\ S,Q) \equiv I\,[E1..E1] \wedge \pi + (C2-C1+1)K + \Delta_{forst} = C$$

The routine call is another candidate for simplification. It is often desirable to think of routines in the same way as the primitive operators of the language, but the general cost predicates treat routine calls and operators differently. If the cost of the routine is constant and its post condition indicates this, then we can introduce a new constant Δ_f to represent that cost. More formally,

$$\underline{cp}(\underline{call}\ f, Q) \equiv (P_f \wedge (Q_f \supset Q))_{\pi}^{\pi+\Delta_f}$$

where $\Delta_f = K + \Delta_{call}$

provided the specifications of f are $P_f \wedge \pi{=}C$ {body of f} $Q_f \wedge \pi{=}C{+}K$

(P_f and Q_f are assumed not to involve π)

When the theory is extended to deal with parameters to routines, this simplified rule can be similarly extended. The cost of a routine will often depend on some of its explicit or implicit parameters; this can be handled by treating Δ_f as a function of those parameters.

6. Discussion

We have presented a formal system for verifying assertions about program performance. This system draws heavily on ideas from both algorithmic analysis and program verification. It formalizes the operation-counting methodology of algorithmic analysis, and it is compatible with present verification techniques based on predicate transformers. We consider each of those relationships and some of the remaining issues below.

6.1. Relation to Algorithmic Analysis

As a formalization of the operation-counting methodology, the system presented here retains many of the properties of that approach. Both assume that operation counts adequately characterize costs, and both require the human anaylzer to provide the insight that leads to the results. In addition, the cost predicate formalism specializes to several common cost analysis strategies; by offering a common explanation of these strategies it helps to unify them.

We, like the algorithmic analysts, have assumed that all costs can be attributed to discrete, identifiable events and that these costs are context-independent. Neither assumption is quite accurate in practice. Distributed costs such as those imposed by multiprocessing overhead and distributed savings such as those provided by caches or optimizing compilers are inadequately modelled. It is important to learn how much error is introduced by these simplifying assumptions, but it is clear that good estimates (more accurate than simple order-of-magnitude statements) are useful in practice, even if they are not exact. Moreover, the field of algorithmic analysis has not foundered on these same assumptions.

It is important to observe that the cost predicates support verification of assertions about performance, *not* derivation of costs. True, they can be used to derive costs in simple cases (e.g., no loops and simple cost equations). In general, however, the human user must provide knowledge about the program by supplying loop invariants and managing the analysis of inequalities.

6.2. Relation to Program Verification

The influence of traditional program verification issues on the cost predicate system extends beyond the obvious similarities in the form of the rules. The decision to base the system on a new sort of predicate transformer was prompted by the utility of predicates involving the program variables for maintaining information about data values. Since predicate transformers (of any sort) can only deal with explicit state, we were led in turn to add the resource cost to the program state. As a result, accounting for the resource costs does not add new kinds of assertions to the programs or new kinds of processing to the verification task. The happy outcome is that assertions about functionality and about performance are quite compatible, and verification of both kinds can proceed simultaneously.

Just as the set of weakest precondition rules for a language is a set of predicate transformers that expresses the functional semantics of the language, the set of cost predicates provides a definition of the costs that will be incurred by a program.[1] Indeed, decisions about which costs to include and where to charge them are simply decisions about the expected meaning (i.e., costs) of the language. The cost predicates could be rewritten, for example to describe a different object machine architecture.

6.3. Extensions

This paper has presented a basic description of a system for dealing with properties of programs that cannot be described purely in terms of program variables. We have addressed execution time specifically, but the success for that domain suggests similar efforts for other resources and also indicates that other kinds of predicate transformers may provide an expressive medium for other kinds of reasoning about programs.

Specific follow-up tasks for the system presented here include

- validation against a real compiler,
- construction of a similar set of predicate transformers for verifying assertions about utilization of space and other resources,
- exploration of the simplifications of section 5, particularly the problems of

[1] And just as a compiler may not provide exactly the functionality described by the weakest preconditions, it may also fail to provide exactly the efficiency described by the cost predicates. This is a matter of concern only to the extent that the efficiency estimates provided are insufficiently accurate.

dealing with expected values and order-of-magnitude extimates,[2]

- application to data abstraction languages such as Alphard, using the ability to associate assertions with types to modularize the analysis of efficiency as well as functionality, and

- incorporation of these rules in a verification system.

Acknowledgements

This work has profitted from many discussions with members of the CMU Computer Science Department and the ARPA Quality Software Working Group. Suggestions from Bill Wulf, Jim Horning, Butler Lampson, and Jon Bentley have been particularly helpful.

References

[Aho74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Borodin75] A. Borodin and I. Munro, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, 1975.

[Booth79] Taylor L. Booth, "Performance Optimization of Software Systems Processing Information Sequences Modeled by Probabilistic Languages", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1, January 1979, pp. 31-44.

[Dijkstra76] Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[Hoare72] C.A.R. Hoare and Niklaus Wirth, "An Axiomatic Definition of the Programming Laguage PASCAL", *Acta Informatica* 2, (1973), pp. 335-355.

[Knuth73] Donald E. Knuth, *The Art of Computer Programming, vol 1: Fundamental* Algorithms, Addison-Wesley, 1973.

[Knuth69] Donald E. Knuth, *The Art of Computer Programming, vol 2: Seminumerical* Algorithms, Addison-Wesley, 1969.

[Manna74] Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, 1974.

[Manna77] Zohar Manna and Richard Waldinger, *Studies in Automatic Programming* Logic, Elsevier North-Holland, 1977.

[Wegbreit76] Ben Wegbreit, "Verifying Program Performance", *Journal of the ACM*, Vol. 23, No. 4, October 1976, pp. 691-699.

[Wulf80] William A. Wulf, Mary Shaw, Lawrence Flon, and Paul N. Hilfinger, *Fundamental* Structures of Computer Science, to be published by Addison-Wesley, 1980.

---

[2]The problem of converting a probabilistic description of program input to estimates of loop execution frequency was addressed in [Wegbreit76], and the use of probabilistic grammars for estimating the frequencies of calls on operations was presented in [Booth79].