Axiomatic Analysis of Programs and Program Schemes

Thomas J. Ostrand

D.C.S. Technical Report 61

Department of Computer Science
Livingston College, Rutgers University
New Brunswick, New Jersey 08903

September 1977
Axiomatic Analysis of Programs and Program Schemes

ABSTRACT

Axiomatic analysis extracts from the functions and predicates of a program the properties and relations upon which the correct functioning of the program is based. The analysis is done by converting the program to a program scheme and studying a correctness proof for the scheme.

Information of this sort can be used to look for programs whose correctness does not depend on axioms of arithmetic which may not hold in a computer implementation. Another application is to the problem of choosing appropriate data structures for implementation of an algorithm.

Several sample programs are examined, and in one case, a comparison made of the properties required for implementations of different recursive definitions of a function.
1. Introduction

The main steps involved in verifying the correctness of a program are attaching appropriate assertions which describe the state of the computation to various points in the flowgraph of the program, generating verification conditions which relate the state at one point and the ensuing computation to the state at a following point, and finally, proving that the verification conditions are valid statements of the language in which the program is written.

The truth or falsity of the verification conditions depends, in general, on two types of facts. There is first, information about the meaning of the calculations carried out by the program. An example of such information is the fact that \( i \cdot y = y + (i-1) \cdot y \). This identity might be used to show the correctness of a program which performs multiplication by repeated addition.

Second, there is information which is dependent solely on the syntactic form or structure of the program, such as the fact that function \( f \) is always applied twice to a particular variable in a particular assignment statement. This type of information contributes to the correctness of the program independently of the meaning of the function \( f \).

We call the first type of information semantic or domain-related information, and the second type we call syntactic or control-related. It is useful to try separating the parts of a proof which depend on these two types of facts, since the control-related information is relevant to the correctness of any
program which has the same syntactic form as the original program, but possibly different meanings for its functions and predicates.

Studying proof techniques for program schemes is an obvious way to do this separation, and we discuss in this paper the analysis of program schemes with the goal of discovering which properties of the interpreted symbols contribute to the correctness of programs based on the scheme.

It has long been recognized that standard correctness proofs of many programs are, in fact, not valid, since the proofs usually ignore such common computer phenomena as overflow, round-off error, truncation, and other deviations from the properties of idealized arithmetic. Since axiomatic analysis reveals which properties are being assumed for a correctness proof, we can use this technique to look for programs whose correctness does not depend on possibly unfulfilled axioms of arithmetic. If such a program cannot be found for a particular problem, we at least will be explicitly aware of possible sources of errors in a "verified" program.

Hoare (1969) has noted that the numbers of a computer arithmetic are a finite set, and that there are differing ways of implementing the overflow situation which occurs when an operation produces a result beyond the limit of this set. He suggests three possible independent axioms, from which to choose one, according to the given implementation. With axiomatic analysis we are approaching this implementation problem from the opposite direction, asking first which properties actually need
be assumed.

2. Method

For a given program scheme $S$, one can write input and output assertions $\Phi$ and $\Psi$ in the language of the scheme, and define a notion of correctness of $S$ with regard to $\Phi$ and $\Psi$. A correctness proof can be carried out in essentially the same way as for a program, namely by attaching intermediate assertions to cutpoints of the scheme, and showing that all path verification conditions are valid statements. So long as no special conditions or properties have been assumed, we may then conclude that any interpretation of $P$ is a correct program with respect to the corresponding interpretations of $\Phi$ and $\Psi$.

It is clear that there exist many programs whose correctness is not an instance of the correctness of a scheme in the above sense. One cause for this is the use of semantic information in the computation method of a program. Another cause, which is quite similar, is the occurrence in the output assertion of a function or predicate which does not appear in the program itself. For instance, the previously mentioned program computes the product of an integer and another number by repeated addition. Obviously, there is no occurrence of the multiplication function within the program, yet the statement of its correctness must mention this function.

In both these instances, it is still possible to verify a schematized version of the program, and then conclude that the
program and its assertions are instances of the scheme and its assertions. However, it is necessary to add to the purely syntactic scheme a description of the semantic properties which guarantee the program's correctness. Such descriptions are provided either as equations between two syntactic terms in the scheme's language, or else as logical statements in the first-order language whose terms are the terms of the scheme's language.

Both the equations and the logical statements are to be understood as axioms for showing the correctness of the scheme. Any variable which occurs in an axiom may be replaced uniformly through the axiom by any term in the scheme's language. The effect of using axioms to verify a scheme is to restrict the set of possible interpretations of the scheme to those which satisfy the axioms. The situation is analogous to the axiomatic study of various areas of mathematics. Any theorem proved about abstract groups is true of any real system which satisfies the group axioms. If one wishes to prove a theorem which requires the assumption of commutativity, then an appropriate axiom can be added to the basic group axioms. The theorem, of course, only holds for abelian groups; that is, the set of possible interpretations of the group axioms has been restricted to those which additionally satisfy commutativity.

Our basic goal here is to discover axioms which are sufficient to verify schematizations of programs. In this way we hope to gain knowledge about the conditions which really are at work in making an algorithm correct.
Our basic technique is to attempt a proof of the schematization of a program in exactly the same way as the proof of the program was carried out. Whenever a verification condition for the scheme is not a universally valid sentence, we introduce one or more appropriate axioms which enable the proof to go through. The set of axioms required to prove all the verification conditions then characterizes the interpretations of the scheme which can be considered correct. Of course, we only introduce axioms which are indeed true for the interpretation which is the original program.

3. Examples

Example 1: Integer part of square root

We show in Figure 1 the flowcharts of both the program and its translation into a scheme. The program calculates in $u$ the largest integer less than or equal to the square root of the input value $x$. The method is based on the fact that the sum of the first $i$ odd integers is $i$ squared.
Program P1

Figure 1

We identify the three points in the program where assertions are made:

1) the input assertion $\mathcal{P}(x): x \geq 0$
2) the loop assertion: $u^2 \leq x$ & $v=(u+1)^2$ & $w=2u+1$.
3) the output assertion $\mathcal{Y}(x, z): z^2 \leq x < (z+1)^2$
To schematize these assertions requires the introduction of two function symbols and one predicate symbol which do not appear in the scheme itself. The function symbols are \( S(x) \), representing \( x^2 \) and \( D(x) \), representing \( 2x \). The predicate symbol is \( T(x,y) \), representing \( x \geq y \).

The assertions for the scheme can now be written as follows:
The input assertion \( A_1: \quad T(x,a) \)
The loop assertion \( A_2: \quad T(x, Su) \land v = S(gu) \land w = g(Du) \).
The output assertion \( A_3: \quad T(x, Sz) \land P(S(gz), x) \).

The verification conditions are

\[ V_12: \quad T(X,a) \Rightarrow [T(X, Sa) \land a = S(ga) \land b = g(Da)] \]
\[ V_22: \quad [T(x, Su) \land u = S(gu) \land w = g(Du) \land p(v, x)] \Rightarrow [T(x, Sgu) \land f(v, hw) = S(ggu) \land hw = g(Dgu)] \]
\[ V_23: \quad [T(x, Su) \land v = S(gu) \land w = g(Du) \land p(v, x)] \Rightarrow [T(x, Su) \land p(S(gu), x)] \]

\( V_{23} \) is immediately seen to be true by substitution. The other two conditions require introduction of the following axioms.
Axiom

1. \(-p(x,y) \Rightarrow T(y,x)\)
2. \(g(hx)=h(gx)\)
3. \(h(Dx)=D(gx)\)
4. \(S(gx)=f(Sx,g(Dx))\)
5. \(f(a,x)=x\)
6. \(ga=b\)
7. \(Sa=a\)
8. \(Sb=b\)
9. \(Da=a\)

Interpretation to P1

- \(-x > y \Rightarrow y = x\)
- \((x+2)+1 = (x+1)+2\)
- \(2x+2 = 2(x+1)\)
- \((x+1)^2 = x^2 + (2x+1)\)
- \(0+x = x\)
- \(0+1 = 1\)
- \(0^2 = 0\)
- \(1^2 = 1\)
- \(2*0 = 0\)

Axioms 5–9 are needed to prove the first verification condition; they state special properties of the constant symbols with respect to the functions. We now show the proof of each conjunct in the conclusion of V2.

First: \(-p(v,x)\) hypothesis

\[ T(x,v) \quad \text{Axiom 1} \]
\[ T(x,SGu) \quad \text{substitution} \]

Second: \(S(ggu)=f(Sgu,g(Dgy))\) Ax. 4

\[ =f(v,g(hDu)) \quad \text{substitution and Ax. 3} \]
\[ =f(v,h(gDu)) \quad \text{Ax. 2} \]
\[ =f(v,h(w)) \quad \text{substitution} \]

Third: \(h(w)=h(gDu)\) substitution

\[ =g(hDu) \quad \text{Ax. 2} \]
=g(Dgu)  Ax. 3

Note that the result of the third conjunct is actually a part of the proof of the second.

Example 2: Factorial and List Reverse

The original program of this example, shown in Figure 2, computes the factorial function. After schematizing it and verifying the scheme's correctness, we see that another interpretation yields the reverse function, operating on lists.

![Diagram](image1)

Program P2

![Diagram](image2)

Scheme S2

Figure 2
The factorial program operates over the non-negative integers; there is no explicit input assertion. The output assertion is simply z=x! The loop assertion is y!z=x!. It is straightforward to show the correctness of the three verification conditions for the program.

Turning to the scheme, we find it necessary to introduce an associativity axiom and an identity axiom for the operation c. Further, we need two facts about the function allegedly computed by the scheme. These are summarized below.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Interpretation to P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. c(x, c(y, z)) = c(c(x, y), z)</td>
<td>x*(y<em>z) = (x</em>y)*z</td>
</tr>
<tr>
<td>2. c(x, a) = c(a, x) = x</td>
<td>x*1 = 1 * x = x</td>
</tr>
<tr>
<td>3. s y P y = a</td>
<td>y = 1 P y = 1</td>
</tr>
<tr>
<td>4. ~s y P F y = c(F(ty), hy)</td>
<td>~y = y P y = (y - 1)! * y</td>
</tr>
</tbody>
</table>

Axioms 3 and 4 could be taken as a definition for the function F, or as a recursion scheme for F. In fact, another way to view the process of verifying that the scheme S2 computes F is to think of it as a demonstration of the equivalence of S2 to the recursion scheme given by Axioms 3 and 4.

The schematized loop assertion for S2 is c(Fy, z) = Fx, and the three verification conditions are:

V12: c(Fx, a) = Fx
V22: c(Fy, z) = Fx & ~s y P (Fty, c(hy, z)) = Fx
V23: c(Fy, z) = Fx & s y P z = Fx.
These are all easily shown with use of the axioms. The list reverse interpretation is a result of applying the following interpretation to the schema.

<table>
<thead>
<tr>
<th>Scheme Syntax</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>lists of elements over some alphabet</td>
</tr>
<tr>
<td>the constant a</td>
<td>( \lambda ), the empty list</td>
</tr>
<tr>
<td>predicate s(x)</td>
<td>( x = \lambda )</td>
</tr>
<tr>
<td>monadic function hx</td>
<td>head ((x))</td>
</tr>
<tr>
<td>monadic function tx</td>
<td>tail ((x))</td>
</tr>
<tr>
<td>dyadic function c(x,y)</td>
<td>( x \sim y ).</td>
</tr>
</tbody>
</table>

The four axioms all hold for this interpretation, and thus the interpreted version does compute the reverse of its input.

**Example 3: Exponentiation**

For the final example we examine several methods of calculating \( A^B \). We note which properties are used to verify each program individually, and then compare three different recursive definitions of exponentiation.

The two programs are, respectively, a traditional iterative exponentiation routine which simply multiplies \( A \) by itself \( B-1 \) times, and a fast program which requires only \( \log B \) multiplications. The second routine has become a popular example for verification studies since its use as an example in King 1969. The programs are shown in Figure 3, as P3 and P4.
Traditional Exponentiation
P3

Fast Exponentiation
P4

Figure 3
To verify these two programs, we use the following recursive definition of exponentiation.

\[
\exp(A,B) = \begin{cases} 
1 & \text{if } B = 0 \\
A \cdot \exp(A,B-1) & \text{if } B > 0 
\end{cases}
\] (1)

This would probably be considered the most natural definition of exponentiation. We examine later the consequences of using a different definition in carrying out the verifications.

Despite the difference in calculation method, the two programs share the same loop invariant assertion. However, the verification conditions for the loop paths are different, reflecting the assignments made within the loop bodies. The assertions are:

\( \varphi(A,B): \text{true} \)

loop assertion: \( z \cdot \exp(x,y) = \exp(A,B) \)

\( \psi'(A,B,z): z = \exp(A,B) \)

Program schemes resulting from P3 and P4 are shown as S3 and S4 in Figure 4. The assertions are schematizations of the above program assertions. The output function for the schemes is the following abstraction of the recursive definition of exponentiation:

\[
F(x,y) = \begin{cases} 
c & \text{if } py \\
g(x,F(x,dy)) & \text{if } \neg py 
\end{cases}
\] (2)
Schematization S3 of P3

Schematization S4 of P4

Assertions
\[ \varphi(A, B): \text{true} \]
\[ \text{loop: } g(z, F(x, y)) = F(A, B) \]
\[ \psi(A, B, z): z = F(A, B) \]

Figure 4
The correspondence between abstract symbols appearing in the schemes and actual functions and predicates is summarized in Figure 5.

<table>
<thead>
<tr>
<th>Abstract Symbol</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>predicate px</td>
<td>x = 0</td>
</tr>
<tr>
<td>predicate sx</td>
<td>x = 1</td>
</tr>
<tr>
<td>function fx</td>
<td>x\mod 2</td>
</tr>
<tr>
<td>function hx</td>
<td>x/2 (integer division)</td>
</tr>
<tr>
<td>function g(x,y)</td>
<td>x*y</td>
</tr>
<tr>
<td>constant c</td>
<td>1</td>
</tr>
</tbody>
</table>

Interpretations of Symbols for \( \exp(A,B) \)

Figure 5

We first examine the proof of S3 to determine which axioms are needed. The verification conditions are:

V12: \( g(c,F(A,B)) = F(A,B) \)

V22: \( g(z,F(x,y)) = F(A,B) \) & \( -py \supset g(g(z,x),F(x,dy)) = F(A,B) \)

V23: \( g(z,F(x,y)) = F(A,B) \) & \( py \supset z = F(A,B) \)

Proofs of all three of these are straightforward, and require only the following two axioms, which are merely statements of elementary properties of multiplication.
Axiom                        Interpretation
1. \( g(c,x) = g(x,c) = x \)   1\( x = x \cdot 1 = x \)
2. \( g(x, g(y,z)) = g(g(x,y),z) \) \( x \cdot (y \cdot z) = (x \cdot y) \cdot z \)

The simplicity of proof of these verification conditions can be explained by noting that the program is almost a mechanical implementation of the recursive definition. In fact, the only real change made in the program is that the multiplications are accumulated in the opposite order; this is what necessitates use of the associativity axiom.

In contrast, the efficient program is based on an entirely different algorithm for computing exponentiation, and makes use of functions which do not appear in the recursive definition. The proofs will require axioms relating the symbols for these new functions to the decrement function symbol. The main application of these axioms is in the proof of a lemma about the function symbol \( F \); the lemma serves to connect the definition of \( F \) in terms of the symbol \( d \) (representing decrement by 1), with the operation of the program in terms of \( h \) (representing integer division by 2.).

\[
F(x,y) = \begin{cases} 
F(g(x,x),hy) & \text{if pfy} \\
\text{pfy} & \text{if sfy} \\
g(x,F(g(x,x),hy)) & \text{if sfy} 
\end{cases}
\]

One of the axioms for the verification of \( S4 \) is \( \text{pfy} \iff \text{sfy} \); thus these two conditions are mutually exclusive and exhaustive.
The lemma's proof is delayed until after the verification conditions for $S_4$ have been discussed.

Conditions V12 and V23 for $S_4$ are identical to those for $S_3$; their proofs require only Axiom 1. Since the loop body of $S_4$ consists of two paths, we generate two conditions. They are

$$V22': \quad g(z,F(x,y)) = F(A,B) \wedge \neg py \wedge \neg sfy \quad \Rightarrow \quad g(z,F(g(x,x),hy)) = F(A,B)$$

$$V22'': \quad g(z,F(x,y)) = F(A,B) \wedge \neg py \wedge sfy \quad \Rightarrow \quad g(g(z,x),F(g(x,x),hy)) = F(A,B)$$

With the introduction of Axiom 3: $py \iff \neg sfy$ and the use of the lemma, proofs of both these conditions are simply a matter of substitution and in the case of $V22''$, one application of Axiom 2.

We now turn to the proof of the lemma, which will provide most of the axioms which are ultimately behind the correctness of program $P_4$.

In proving the lemma we make use of an abstract induction principle which is valid for any set of objects on which a well-founded partial ordering is defined. The induction principle will be applied to the second argument of $F$, for which the concrete interpretation is the set of non-negative integers. The principle is presented here in a form which is especially suitable for use in proofs of equivalences among program and recursion schemes.

**Induction Principle:** Given a set $S$, mapping $d:S \rightarrow S$, and predicate $p:S \rightarrow \{\text{true, false}\}$, with the property that for each
x ∈ S, there is an integer i called the height of x, such that p(d^ix) = true.

Let Q be a subset of S such that

1. if x ∈ S and px holds, then x ∈ Q
2. if x ∈ S and dx ∈ Q, then x ∈ Q

Then Q = S.

The principle is easily proved by mathematical induction on the height of elements of S.

To prove the lemma, we assume that the domain of the second argument of F(x,y) is a well-founded set S of the type described in the Induction Principle. Let Q be the subset of S for which (3) holds.

First, let y ∈ S and py hold. We introduce Axiom 4:
py ⊃ phy and Axiom 5: py ⊃ pfx. Then, from the definition of F, F(x,y) = c and F(g(x,x),hy) = c. Since pfx holds, (3) is satisfied, and y ∈ Q.

For the induction step, let y ∈ S, ¬py, and dy ∈ Q. We consider two cases.

1. pfx holds

Then F(x,y) = g(x,F(x,dy)) since ¬py
   = g(x,g(x,F(g(x,x),hdy))) Axiom 6 and Ind. Hyp.
   = g(g(x,x),F(g(x,x),hdy)) Axiom 2
   = g(g(x,x),F(g(x,x),dhy)) Axiom 7
   = F(g(x,x),hy) Axiom 8 and defn F
2. sfy holds

Then $F(x, y) = g(x, F(x, dy))$ since \(\neg py\)

\[
= g(x, F(g(x, x), hdy)) \quad \text{Axiom 9 and Ind. Hyp.}
\]

\[
= g(x, F(g(x, x), hy)) \quad \text{Axiom 10}
\]

Thus (3) holds in both cases, \(y \in Q\), and the lemma is proved.

The axioms introduced for the entire verification of the scheme $S_4$ are collected in Figure 6, together with their interpretations.

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $g(c, x) = g(x, c) = x$</td>
<td>$1 \times x = x \times 1 = x$</td>
</tr>
<tr>
<td>2. $g(x, g(y, z)) = g(g(x, y), z)$</td>
<td>$x \times (y \times z) = (x \times y) \times z$</td>
</tr>
<tr>
<td>3. $py \iff \neg sfy$</td>
<td>$y \mod 2 = 0 \iff y \mod 2 \neq 1$</td>
</tr>
<tr>
<td>4. $py \supset phy$</td>
<td>$y = 0 \supset y/2 = 0$</td>
</tr>
<tr>
<td>5. $py \supset pfy$</td>
<td>$y = 0 \supset y \mod 2 = 0$</td>
</tr>
<tr>
<td>6. $pfy \supset sfdy$</td>
<td>$y \mod 2 = 0 \supset (y-1) \mod 2 = 1$</td>
</tr>
<tr>
<td>7. $pfy \supset hdy \supset dhy$</td>
<td>$y \mod 2 = 0 \supset (y-1)/2 = y/2 - 1$</td>
</tr>
<tr>
<td>8. $\neg py \land pfy \supset \neg phy$</td>
<td>$y \neq 0 \land y \mod 2 = 0 \supset y/2 \neq 0$</td>
</tr>
<tr>
<td>9. $sfy \supset pfdy$</td>
<td>$y \mod 2 = 1 \supset (y-1) \mod 2 = 0$</td>
</tr>
<tr>
<td>10. $sfy \supset hdy = hy$</td>
<td>$y \mod 2 = 1 \supset (y-1)/2 = y/2$</td>
</tr>
</tbody>
</table>

Axioms for $S_4$ Verification

Figure 6
4. Equivalence Between Recursion Schemes

As mentioned in connection with Example 2, the verification of the two preceding program schemes can be regarded also as demonstrations of equivalence between a program scheme and a recursion scheme.

Another means of studying the properties used by particular implementations of a function is to prove equivalence between purely recursive versions which purport to compute the same function. There are at least two alternative recursive formulations of the exponentiation function, in addition to (1). Both are closely related to the efficient program P4, but the first one is the naturally corresponding recursive definition for P4.

\[
\text{exp1}(A, B) = \begin{cases} 
1 & \text{if } B = 0 \\
\text{exp1}(A^2 A, B/2) & \text{if } B \mod 2 = 0 \text{ and } B > 0 \\
A \cdot \text{exp1}(A^2 A, B/2) & \text{if } B \mod 2 = 1
\end{cases}
\] (4)

An attempt at creating an iterative implementation of \text{exp1} leads directly to the flowchart P4. An abstract version of \text{exp1} is the following recursion scheme, \text{T}(x,y).

\[
\text{T}(x,y) = \begin{cases} 
c & \text{if } py \\
\text{T}(g(x,x),hy) & \text{if } pfy \land \neg py \\
g(x,\text{T}(g(x,x),hy)) & \text{if } sfy \land \neg py
\end{cases}
\] (5)

It is immediately apparent that (5) is almost identical to the lemma which provided the keystone of the correctness proof of program P4. The lemma can also be used to prove the equivalence (subject to Axioms 1-9) of recursion schemes (2) and (5).
The third recursive formulation of exponentiation is what might be created in a first attempt to utilize the idea of breaking down the exponent of \( A^B \) according to its binary number representation.

\[
\exp_2(A,B) = \begin{cases} 
1 & \text{if } B = 0 \\
A & \text{if } B = 1 \\
\exp_2(A,B \mod 2) \times \exp_2(A,B/2) \times \exp_2(A,B/2) & \text{if } B > 1 
\end{cases}
\]  
(6)

Here we see an expression of the idea that the exponent \( B \) can be divided by 2, and the final result expressed in terms of the rightmost bit of \( B \) and the remaining bits. The conversion of (6) into a scheme results in the following form.

\[
R(x,y) = \begin{cases} 
\circ & \text{if } py \\
x & \text{if } s\overline{y} \& \overline{py} \\
g(R(x,y), g(R(x,\overline{y}), R(x,\overline{y}))) & \text{if } \overline{py} \& \overline{s\overline{y}} 
\end{cases}
\]  
(7)

Since (6) is a correct definition of the exponentiation function (its equivalence to both (1) and (4) can be shown by recursion induction (McCarthy 1963) or structural induction (Burstall 1969)), one expects that the program schemes \( S3 \) and \( S4 \) can be verified using \( z = R(x,y) \) as the output assertion. This is indeed correct, and in Figure 7 are shown the axioms necessary to verify \( S4 \). These may be contrasted with the axioms of Figure 6. The major difference is the use of commutativity of multiplication in the proof using \( R(x,y) \) as output function. Evidently many of the special facts about the \( \text{mod}2 \) and integer division functions, which we required in the original proof, have
been incorporated into the definition (6).

**Axiom**

<table>
<thead>
<tr>
<th>1. pfx ∨ sfx</th>
<th>x mod 2 = 0 ∨ x mod 2 = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. sx ⊃ phx</td>
<td>x = 1 ⊃ x/2 = 0</td>
</tr>
<tr>
<td>3. sx ⊃ sfx</td>
<td>x = 1 ⊃ x mod 2 = 1</td>
</tr>
<tr>
<td>4. g(c,x) = g(x,c) = x</td>
<td>1<em>x = x</em>1 = x</td>
</tr>
<tr>
<td>5. g(x,g(y,z)) = g(g(x,y),z)</td>
<td>x*(y<em>z) = (x</em>y)*z</td>
</tr>
<tr>
<td>6. g(x,y) = g(y,x)</td>
<td>x<em>y = y</em>x</td>
</tr>
</tbody>
</table>

Axioms for S4 Verification with R(x,y) Output Function

Figure 7

5. Summary

Axiomatic analysis of programs and schemes tells us which properties of the data domain of a program are being invoked to guarantee the program's correctness. Such information is useful in discovering whether the program will operate properly despite the irregularities of computer arithmetic. In addition, if the axioms required for correctness are few in number, and another interpretation for the verified scheme can be found, then the verification has gone beyond the original program.

Another application is to the problem of choosing an
appropriate data structure for implementation of an algorithm. Axiomatic analysis of the algorithm as a scheme, with an abstract data structure, should provide information about the properties of the structure required. An appropriate implementation (stack, queue, linked list, etc.) can be chosen on this basis.

These methods also have potential usefulness in studying transformations from recursive definitions to iterative implementations. Through understanding the requirements of a recursive formulation, and the different ones of various non-recursive programs, one can make a more intelligent choice of implementation from among the latter.
References


