Verification of logic programs and imperative programs

Lee Naish

Technical Report 90/6
(Revised May 1991)
Department of Computer Science
University of Melbourne
Parkville, Victoria 3052
Australia
lee@cs.mu.OZ.AU

Abstract

This paper explores the relationship between verification of logic programs and imperative programs with the aim of uncovering the kinds of reasoning used to construct logic programs. We discuss forward reasoning, such as that used for verifying imperative programs using the inductive assertion method, and backward reasoning, such as that used for verifying imperative programs using subgoal induction and logic programs using consequence verification. We argue that consequence verification is often inadequate for Prolog programs because programmers make implicit assumptions about how procedures are called. These assumptions can be made explicit using general type declarations.

Verification of logic programs with type declarations can be done in two steps. We show that one corresponds to subgoal induction and the other corresponds to the inductive assertion method. Thus two existing verification methods are combined. The forward and backward reasoning inherent in this method of verification can be used for constructing Prolog programs. Type declarations can be used to document the equivalent of loop invariants in the inductive assertion method. We discuss how Prolog programs can be constructed using different combinations of forward and backward reasoning. Backward reasoning can result in simpler, reversible procedures. Forward reasoning can result in code which is more efficient or has more intuitive declarative semantics. Code which only uses forward reasoning is also efficient for bottom-up evaluation.

Keywords: program verification, program construction, logic programming, imperative programming, types, bottom-up evaluation, context transformation

1 Introduction

This paper investigates verification of iterative imperative programs and logic programs, and the relationship between them. However, the main purpose is not the investigation of verification methods per se. We are more interested in how programs are constructed. Formal verification methods can give insight into the logic people use, often implicitly, when programming and can lead to more systematic programming methodologies. We will be looking at two styles of reasoning used to construct algorithms — forward reasoning and backward reasoning. Forward reasoning is typified by the construction of iterative algorithms, where the initial conditions are transformed repeatedly until the final answer is computed. Backward reasoning is typified by the construction of recursive algorithms, where the base cases are defined and more complex initial conditions are handled by recursively solving simpler problems and applying a transformation.

Forward and backward reasoning can both arrive at the same algorithm, the same algorithm can be expressed iteratively and recursively and recursion in logic programs can be implemented by top-down and bottom-up evaluation. By studying verification of imperative and logic programs, writing programs in various languages can be better understood. In particular, we believe that writing Prolog programs is not well understood, and writing logic programs which are to be evaluated bottom-up is understood even less well.

One of the advantages of logic programming is that specifications, programs and verification can all be viewed within a single uniform framework. Programs can be verified correct simply by checking that each clause is true in the intended model of the program. This is known as consequence verification. Verification of imperative programs is more cumbersome, since additional theory is needed to relate program statements to logic, and some of the logic of the program (such as loop invariants) is often not explicitly available. However, if loop invariants are given, it can be shown that certain methods of verifying iterative imperative programs are the same as consequence verification of logic programs which implement the same algorithm [CvE81]. Specifically, using subgoal induction is equivalent to verifying logic programs which use top-down evaluation (like Prolog) and using inductive assertions is equivalent to verifying logic programs which use bottom-up evaluation.

For iterative programs, the inductive assertion method corresponds to what we call forward reasoning and the subgoal induction method corresponds to backward reasoning. For Prolog programs, consequence verification corresponds to backward reasoning and for bottom-up logic programs consequence verification corresponds to forward reasoning. The use of forward reasoning to write and verify Prolog programs has been largely ignored. However, work on the relationship between specifications, logic programs and type information suggests that often consequence verification cannot be directly applied to Prolog programs [Nai90]. The problem is that many Prolog programs make implicit assumptions about the types of their arguments. For example, the intended semantics of `append` is that all arguments should be lists, but the with the normal definition, calls can succeed with non-lists. The base case of `append` is false in the intended interpretation, and consequence verification fails.

Despite the existence of a clause which is false in the intended interpretation, if a well typed call to `append` succeeds, it is guaranteed to be correct. When defining `append`, it is assumed that top level calls will be well typed, and this is maintained as an invariant in subsequent recursive calls. More generally, there can be arbitrary assumptions about calls. Maintaining these assumptions as invariants requires forward reasoning.

The type scheme of [Nai90] allows very general assumptions to be expressed explicitly by using type declarations. The assumptions do not cause incorrect answers as long as the program is *type correct*. Using this scheme, verification of Prolog programs can be done in two parts. First, we use forward reasoning to show that the program is type correct. This is similar to using inductive assertions. Second, we use consequence verification, a form of backwards reasoning, to show that the program with extra assumptions is correct. This is equivalent to using subgoal induction. These two verification techniques for imperative programs can also be combined in a similar way.

We argue that typical Prolog programming involves both forward and backward reasoning and that the type scheme provides a way of formalising this. We discuss styles of Prolog programming with different mixes of forward and backward reasoning. Too great a reliance on backward reasoning can lead to inefficient algorithms or predicates with a less intuitive declarative semantics. Too great a reliance on forward reasoning can lead to code which is more complex than necessary and less flexible with respect to the mode of use. However, such programs are still quite efficient when evaluated top-down, and can also be easily transfomed into programs which efficient for bottom-up evaluation.

2 Inductive assertions

A program to calculate the greatest common divisor of two natural numbers is often used as an example of program construction and verification. For imperative programming languages, it illustrates the use of destructive assignment, conditionals and while loops. Below is a segment of Pascal code which implements a common algorithm:

```
1       while x <> y do begin
2               if x > y then
3                       x := x - y
4               else
5                       y := y - x
6       end;
7       answer := x
```

The way this code is typically proved correct is to first introduce two "shadow variables" `x0` and `y0` which are the initial values of `x` and `y` (before line 1). The following loop invariant is then postulated (it is true whenever execution reaches line 1): $\mathtt{gcd}(\mathtt{x},\mathtt{y}) = \mathtt{gcd}(\mathtt{x0},\mathtt{y0})$. Verification of the code involves proving that if the invariant holds then the answer is correct, and proving that the invariant does indeed hold initially (which is trivial in this case) and after each loop iteration. This verification method is know as the *inductive assertion* method [Hoa69, Flo77]; the method known as *predicate transformers* (which uses weakest preconditions) [Dij79, Gri81] results in an equivalent proof of partial correctness but can also be used for proving termination and for deriving the program. The first part of the proof is simple, since we know that the loop will only terminate when $\mathtt{x} = \mathtt{y}$, and $\mathtt{x} = \mathtt{y} \wedge \mathtt{gcd}(\mathtt{x}, \mathtt{y}) = \mathtt{gcd}(\mathtt{x0}, \mathtt{y0})$ implies that $\mathtt{x} = \mathtt{gcd}(\mathtt{x0}, \mathtt{y0})$. Proving that the invariant holds is a little more complex. To show it holds at the end of the loop, we can work backwards and find that it is sufficient that at the start of the loop (between lines 1 and 2) the following holds:

$$(\mathtt{x} > \mathtt{y} \wedge \mathtt{gcd}(\mathtt{x0}, \mathtt{y0}) = \mathtt{gcd}(\mathtt{x} - \mathtt{y}, \mathtt{y})) \vee (\mathtt{x} =< \mathtt{y} \wedge \mathtt{gcd}(\mathtt{x0}, \mathtt{y0}) = \mathtt{gcd}(\mathtt{x}, \mathtt{y} - \mathtt{x}))$$

We need to prove that $\mathtt{gcd}(\mathtt{x}, \mathtt{y}) = \mathtt{gcd}(\mathtt{x0}, \mathtt{y0}) \wedge \mathtt{x} <> \mathtt{y}$ implies this condition, which can be done using knowledge of the gcd function.

3 Consequence verification

There is a close correlation between the logic in a logic program and in the proof of (partial) correctness of an equivalent imperative program. The logic behind the imperative program (loop invariants and the like) is exposed when written as a logic program. The most straightforward coding of the above gcd algorithm into Prolog is as follows:

```
gcd(X, X, X).
gcd(X, Y, Z) :- X > Y, X1 is X - Y, gcd(X1, Y, Z).
gcd(X, Y, Z) :- X < Y, Y1 is Y - X, gcd(X, Y1, Z).
```

The while loop is converted into a recursive procedure. The first (non-recursive) clause implements the exit condition of the while loop. The other two clauses implement the if-then-else (if-then-else could also be used in the Prolog version) and the updating of `x` and `y`. Because of the lack of destructive assignment, more variables are needed. With a good Prolog compiler, this program can result in an execution which is very similar to the Pascal program, with essentially the same sequence of tests, arithmetic and assignments, so it does implement the same algorithm.

To prove that the program is partially correct, it is sufficient to show that each clause is a true statement about the gcd relation or, equivalently, that each clause is a consequence of the specification of gcd. This technique has been called *consequence verification* [CS77, Hog77, Cla84, Kow85]. For example, to prove the second clause is correct, we need to prove that $\mathtt{x} > \mathtt{y} \wedge \mathtt{gcd}(\mathtt{x} - \mathtt{y}, \mathtt{y}) = \mathtt{z}$ implies $\mathtt{gcd}(\mathtt{x}, \mathtt{y}) = \mathtt{z}$. At this point, there seems little in common between the verification of imperative and logic programs. The imperative program uses the shadow variables `x0` and `y0` in a loop invariant, whereas the logic program never refers to the top level (initial) values of `X` and `Y` and no invariant is needed. Also, the details of the proof are very different. The implications proved for the logic program are almost the converse of those proved for the imperative program.

4 Subgoal induction and top-down evaluation

Even when iteration and recursion result in the same algorithm, they are normally thought of differently. As mentioned previously, iterative programs are normally derived using forward reasoning, whereas recursive programs are normally derived using backward reasoning. However, it is possible to think about iterative programs in a backwards way also. Reasoning about the correctness of the Pascal gcd program in a backwards way leads to a different loop invariant: $\texttt{gcd}(\texttt{x},\texttt{y}) = \texttt{Answer}$, where `Answer` is the value that `answer` will eventually take (this invariant could have been used to derive the program also). Rather than using the value a variable had in the past, we use the value a variable will have in the future. This might seem strange to a Pascal programmer but should be perfectly natural to a Prolog programmer.

In fact, there is a less well-known verification method for imperative programs called *subgoal induction* [MW77] which uses precisely this form of loop invariant. Rather than proving the invariant holds initially and showing that if it holds after N iterations of the loop then it holds after N+1 iterations, the induction is reversed. Subgoal induction proves the invariant holds when the loop exits and shows that if it holds after N+1 iterations of the loop then it must have held after N iterations. This effectively reverses the implication we have to prove for the imperative gcd program, making it the same as consequence verification of the Prolog version. The relationship between subgoal induction and consequence verification can be made very clear by considering verification of the following generic Pascal while loop (taken from [Top77]; an equivalent *flow graph* is discussed in [CvE81]):

```
x := f(x);
while not p(x) do
        x := g(x);
x := h(x)
```

The task is to verify that this code computes some relation $\texttt{r}(\texttt{X0},\texttt{X})$, where `X0` is the initial value of `x` and `X` is its final value. To do so, we introduce a loop invariant, $\texttt{q}(\texttt{X1},\texttt{X})$, where `X1` represents the current value of `x` and `X` represents it value at the end of this program segment (after `h` has been called). To verify the partial correctness of the program using subgoal induction we must prove three things.

(M1) The invariant holds at the loop exit (when `p(x)` is true).

(M2) If the invariant holds at the start of iteration $N+1$ then it holds at the start of iteration N (when `p(x)` is false and before `x` has been replaced by `g(x)`).

(M3) If the invariant holds at the start of the loop (after `f` has been applied to the initial value) then the program segment computes `r`.

Using relational notation instead of functional notation (`f(x,y)` instead of `y=f(x)`) for `f`, `g` and `h`, we can spell out these three implications explicitly as Prolog clauses (equivalent logic is given in [Top77] and [CvE81])).

```
r(X0, X) :- f(X0, X1), q(X1, X).                    % M3
```

```
q(X0, X) :- p(X0), h(X0, X).                        % M1
q(X0, X) :- not p(X0), g(X0, X1), q(X1, X).         % M2
```

This Prolog program implements the same algorithm as the Pascal program. The predicate `r` first calls `f`, as in the Pascal version, then calls the recursive predicate `q`. `q` implements the while loop and also incorporates the call to `h` when the loop exits (the call to `h` could just as easily be put in the clause for `r`). Consequence verification of this program involves proving each clause in a consequence of the specification (or is true in the intended interpretation), thus it is identical to using subgoal induction for the equivalent imperative program.

5 Inductive assertions and bottom-up evaluation

It is also informative to perform the analogous transformation using inductive assertions. Using inductive assertions, the following implications need to be proved in order to verify the generic Pascal program [Top77, CvE81]. Here `i(X0,X1)` is the loop invariant, where `X0` represents the initial value of `x` and `X1` represents its current value.

(F1) The invariant holds at start of the loop (after `f` has been applied to the initial value).

(F2) If the invariant holds at the start of iteration N (when `p(x)` is false) then it holds at the start of iteration $N + 1$ (after `x` has been replaced by `g(x)`).

(F3) If the invariant holds at the end of the loop (when `p(x)` is true, and before `h(x)` is called) then the program segment computes `r`.

```
r(X0, X) :- i(X0, X1), p(X1), h(X1, X).             % F3

i(X0, X) :- f(X0, X).                               % F1
i(X0, X) :- i(X0, X1), not p(X1), g(X1, X).         % F2
```

The call to `f` has migrated to the base case of the recursive predicate. However, because `i` is left recursive instead of right recursive, this corresponds to the start of the loop rather than the end. Also, the positive call to `p` has moved out of the recursive loop and into the clause for `r`. As a Prolog program this code is very inefficient. Searching is needed to find the solution (in fact, the search space is infinite), whereas the previous coding resulted in a deterministic program (with a good compiler at least).

The logic itself does not specify how it should be used to compute the answer. It is interesting to see how the program behaves using alternative evaluation strategies [Kow80] such as bottom-up rather than top-down (which Prolog uses). Because the first argument of `i` in the recursive call is the same as the argument in the head of the clause (`X0`), we can use a constrained version of bottom-up evaluation. Execution could proceed as follows. The initial value of `X0` is used in the first argument of `f` in the base case of the recursion. `f(X0,X1)` is evaluated to find the value of `X1`, then we infer that `i(X0,X1)` is true. We then check if `p(X1)` is true. If it is, we evaluate `h(X1,X)` and use the first clause to derive the final answer. Alternatively, if `p(X1)` is false then we evaluate `g(X1,X2)` and use the last clause to infer `i(X0,X2)`. The same steps can then be repeated using `X2` instead of `X1`, and so on until `p` eventually fails.

A great deal of work has been done to improve the efficiency of bottom-up evaluation for deductive databases. A common method is to transform the program into an equivalent program with a smaller search space and use naive (or semi-naive) evaluation of the transformed program. For right recursive programs, such as the one above, particularly efficient methods are available. The NRSU transformation [NRSU89] applied to the program above results in an execution very similar to what we have outlined. It seems quite plausible that it will eventually be possible to compile this code in a similar way to the (top-down) Prolog and Pascal versions. Consequence verification for this different style of code will be equivalent to verifying the equivalent imperative program using inductive assertions. For the sake of completeness, we now give a bottom-up version of the gcd program.

```
gcd_bu(X, Y, Z) :- g1(X, Y, X1, Y1), X1 = Y1, Z = X1.

g1(X, Y, X, Y).
g1(X0, Y0, X, Y) :-
        g1(X0, Y0, X1, Y1),
        X1 ≠ Y1,
        (if X1 > Y1 then
                X is X1 - Y1, Y = Y1
        else
                Y is Y1 - X1, X = X1
        ).
```

It is worth comparing this program with the Pascal version. The base case of `g1` corresponds to the initial conditions of the Pascal program, where the two variables have their initial values. The recursive clause for `g1` is equivalent to the while loop in the Pascal version. The recursive call gets the "current" values of the variables, `X1` and `Y1`, checks they are not equal (the condition of the while loop), then calculates the new values, `X` and `Y`, using the `if-then-else`. The top level predicate, `gcd_bu` corresponds to the loop exit (`X1=Y1`), where `X1` is returned as the answer `Z`.

6 Implicit types

Consequence verification, and much of the theory of logic programming, assumes that clauses in typical bug free programs are true in the intended interpretation of the program and are consequences of the specification (if a formal specification exists). This seems a reasonable assumption since, if it was not true, logic programs would produce incorrect answers. However, it is pointed out in [Nai90] it this is typically *not* the case. Even very simple Prolog procedures often make implicit assumptions about how they are called, and particularly about the types of their arguments. The gcd program is a case in point. The intended semantics (we claim) is that all arguments are integers, and all are greater than zero. The first clause of the gcd program implies that `gcd(fred,fred,fred)` is true. This is not a consequence of any reasonable specification and the clause is not true in the intended interpretation. Although the program is quite reasonable in practice, it cannot be verified just by using consequence verification. Careful readers may have noticed the corresponding flaw in the verification of the Pascal program (we discuss this later). However, the problems associated with implicit assumptions are generally more severe in languages which are not strongly typed.

Taking the conventional view of logic programming semantics, the way the gcd program should be written is as follows, where `pos_int(X)` means `X` is a positive integer. This program can easily be proved correct using consequence verification.

```
gcd_c(X, X, X) :- pos_int(X).
gcd_c(X, Y, Z) :- X > Y, X1 is X - Y, gcd_c(X1, Y, Z).
gcd_c(X, Y, Z) :- X < Y, Y1 is Y - X, gcd_c(X, Y1, Z).
```

As a Prolog procedure this is rather inefficient and not very robust. If the programmer uses `gcd_c` in such a way as to ensure that the arguments are positive integers (which is likely to be the case) then the extra tests are redundant and the original coding is preferable. If `gcd_c` can be called with a negative integer (or non-integral floating point number) the extra tests are also pointless since the procedure can loop indefinitely with the base case never being used. The simplest way to write a robust version of this algorithm in Prolog is as follows:

```
gcd_r(X, Y, Z) :- pos_int(X), pos_int(Y), gcd(X, Y, Z).

        % gcd, assuming args are positive integers
gcd(X, X, X).
gcd(X, Y, Z) :- X > Y, X1 is X - Y, gcd(X1, Y, Z).
gcd(X, Y, Z) :- X < Y, Y1 is Y - X, gcd(X, Y1, Z).
```

Here `gcd_r` tests the type of `X` and `Y`, and `gcd` is written with the assumption that it is initially called with correct types. If the conventional view of the semantics of logic programs is used, we would conclude that the programmer's intended interpretation of `gcd` is a relation which includes the standard gcd relation, plus other facts such as `gcd(fred,fred,fred)`. `gcd_r` computes a subset of this relation which is exactly the standard gcd relation.

However, we believe typical programmers do not think quite like this. When writing the `gcd` procedure, they think only of the standard gcd relation, but restrict their attention to certain types (positive integers). The code is written so that well typedness of successful calls is maintained as an invariant. It is simple to show that if a call to `gcd` is well typed and the tests in the second or third clauses succeed then the recursive call to `gcd` is also well typed. Good programmers may also provide a robust interface procedure (`gcd_r`) which checks that the invariant holds initially. Thus, if the programmer was asked if `gcd(-1,-1,-1)` is true, (we claim) the reply would not be "yes"; it would be "no, but that should never occur in practice".

7 Explicit types

There are many advantages in making the intended declarative semantics of logic programs (or the assumptions made) explicit. [Nai90] presents a model of "types" for pure Prolog for precisely this purpose. We now breifly summarise this scheme. Programs can contain *type declarations* for some or all of the predicates. Types can be defined using arbitrary Prolog procedures, which gives far more expressive power than traditional notions of types. The reason why the type terminology is used is that the additional flexibility if often not used, and the scheme subsumes some other type schemes for Prolog.

The type declarations are used to define a non-standard declarative semantics for the program and can also be used to detect errors. The semantics of a program P, with type declarations, is given by a modified version of the program, S, which has explicit type checks added to every clause of procedures with type declarations. The standard meaning of S is assumed to be what the programmer really had in mind, for bug-free programs at least. The additional type checks in S are never executed, so programs can potentially produce answers which are incorrect with respect to S. However, if the program is *type correct*[1] and the top level answer is well typed (consistent with the type declarations in the program) then the answer is correct.

Type correct programs have the property that for each ground clause instance, if the head is well typed and the body succeeds then the body is well typed. Given this propery, it is easy to show that if the top level goal in a ground successful SLD derivation is well typed then each subsequent goal is also well typed. A program can be proven correct in two stages. First we show that each clause *in S* is correct (that is, the program with additional type checks is true in the intended model). Second, we show that the program is type correct (so the computed answers are correct, even though the type checks were not executed).

To fully express the intentions of the programmer (and to help with documentation and verification) the coding of the `gcd` procedure should be augmented type declaration such as the following:

```
?- gcd(X,Y,Z) type pos_int(X), pos_int(Y).
```

This declaration does not change the procedural semantics of the program (what is actually executed). However, it does change the declarative semantics of the program, by adding the tests to each clause of the `gcd` procedure in S. Consequence verification can be used to show that S is correct. The base case caused problems previously but now has extra tests in the body which constrain `X` to be a positive integer. It is equivalent to the base case in the definition of `gcd_c`. To show that the computed answers are correct, we must also show that the program is type correct.

[1] *Weak* type correctness is sufficient for the purposes of this paper; type correctness also preserves finite failure.

8 Type declarations and preconditions

Type declarations are similar to *preconditions* of imperative programs. A precondition is an assertion that the input of a procedure will be restricted in some way. The restriction should guarantee that the procedure will produce the correct output. That is, the input-output relation computed by the procedure is the relation which is specified. A procedure is correct if the preconditions of procedures it calls are always satisfied and the following holds:

$$(\texttt{precond(I)} \land \texttt{proc(I,O)}) \Rightarrow \texttt{spec(I,O)}$$

The type scheme defines a declarative semantics for a procedure which includes the type information, or precondition. For type correct programs, the definition of a predicate in S captures to the left hand side of the implication above and consequence verification of the definition is essentially the same as proving the implication. The left side of the implication corresponds to the declarative semantics of the program and the right side corresponds to the intended semantics.

Proving that a logic program is type correct is similar to proving that the preconditions of an imperative program are always satisfied. However, there is an important difference. Preconditions rely on fixed input-output modes and a fixed execution strategy. Type correctness is a purely declarative notion and does not rely on any particular input-output modes or execution strategy. A type correct logic program is guaranteed to produce answers which are correct with respect to S with any execution mechanism which is sound with respect to P: left to right or right to left, top-down or bottom-up, sequential or parallel, and any combination of the above. This is particularly important for Prolog programs which contain procedures which are called in several different modes. Later we discuss the type declarations for procedures which are to be used for bottom-up evaluation also.

Other schemes for adding assertions to Prolog have been proposed [CD88] [DM88]. These schemes are more similar to preconditions in that they rely on the top-down, left to right procedural semantics. The assertions can express information, such as the instantiation state of calls, which cannot be expressed in the type scheme. This makes the schemes more powerful, but the lack of a purely declarative framework and reliance on the standard Prolog execution strategy could be seen as disadvantages.

9 Type correctness and inductive assertions

To show that the `gcd` procedure is type correct, we must show that for each ground clause instance with positive integers as the first two head arguments, if the body succeeds then it is well typed. For the first clause this is trivial, since the body is empty. For the body of the second clause to succeed, `X1` must be a positive integer and hence the recursive call to `gcd` is well typed. Similar reasoning can be applied to show that the third clause it type correct. Thus, although `gcd` has a clause which is wrong with respect to the intended semantics, any computed answer in which the first two arguments are positive integers must be correct. The first two arguments can be checked at some higher level within the program, such as the definition of `gcd_r`. Alternatively, if `gcd` was called directly from the top level of the Prolog system, extra type checks could automatically added to the goal to eliminate ill-typed (incorrect) answers.

The gcd_r procedure has no type declaration so any instance of the clause head is considered well typed. To prove it is type correct we must prove that any instance of the body which succeeds is well typed. This is ensured by the explicit tests. Thus any successful call to gcd_r must be correct.

The proof that gcd_r and gcd are type correct is very similar to an inductive assertion proof of the equivalent imperative program. The original Pascal program we gave can return an incorrect answer if called with two identical negative numbers. If we add an extra test before the while loop which checks that the inputs are positive integers (like gcd_r) then the program works correctly. An alternative would be to add a precondition to the Pascal program. This is like using the typed version of gcd directly instead of calling it via gcd_r. Verifying that x and y are always positive integers using inductive assertions is equivalent to verifying that gcd_r is type correct. We demonstrate this equivalence using generic programs, as before. We first give the Pascal and equivalent Prolog programs.

```
if not t(x) then
        abort;
x := f(x);
while not p(x) do
        x := g(x);
x := h(x);
```

```
 r(X0, X) :- t(X0), f(X0, X1), q(X1, X).          % T3

?- q(X0, X) type q_t(X0).
q(X0, X) :- p(X0), h(X0, X).                      % T1
q(X0, X) :- not p(X0), g(X0, X1), q(X1, X).       % T2
```

The programs have an initial test for incorrect inputs, followed by the same code as before. This time we are not concerned with showing that the program computes the exact function or relation we desire. We just want to show that program computes a subset of the relation defined by the types in the logic program. In general, for each Prolog predicate pred we could have another predicate pred_t which defines the valid types for pred. From the Pascal viewpoint, we can think of procedures and functions as having preconditions. However, so as to concentrate on loop invariants, rather than preconditions, we will only consider a type predicate for q — other predicates and functions are assumed to work correctly for any input.

We also restrict the type declaration for q so it only uses the first argument, which corresponds to the current value of x. In general, a type declaration could use the second argument, which corresponds to the final value. However, this introduces an element of backward reasoning, which makes comparison with inductive assertions more difficult. We will also assume that the invariant used in the inductive assertion loop only uses the current value of the variables. Any reference to the initial value of a variable can be done through the current value of another (possibly dummy) variable, so no expressive power is lost.

To verify type correctness we need to prove the following implications.

(C1) The body of clause (T3) implies that q_t(X1) is true. In the Pascal program, this shows that `t(X0)` implies the loop invariant holds when the loop is reached initially (when `x = f(X0)`).

(C2) The body of clause (T2) and q_t(X0) implies that q_t(X1) is true. In the Pascal program, this shows the invariance of `q_t(X1)` (by the inductive assertion method, not subgoal induction).

```
q_t1(X) :- t(X0), f(X0, X), q(X, _).                          % C1
q_t1(X) :- q_t1(X1), not p(X1), g(X1, X), q(X, _).            % C2
```

We now make one final assumption: if the invariant holds then the loop always terminates successfully. The occurrences of `q(X, _)` can then be deleted. The first occurrence corresponds to the when the loop is entered for the first time; the second corresponds to subsequent iterations. Thus, in order to verify type correctness, we must prove the following:

```
q_t1(X) :- t(X0), f(X0, X).                         % From C1
q_t1(X) :- q_t1(X1), not p(X1), g(X1, X).           % From C2
```

The implications needed to verify the invariant in the Pascal program using inductive assertions can also be simplified. As previously mentioned, we assume the invariant only refers to current values of variables, so the first argument of `i` in (F1) and (F2) can be eliminated. The simplified forms of (F2) and (C2) are equivalent. If we add the extra test `t(X0)`, to (F1) to reflect the extra test in the code, then the simplified forms of (F1) and (C1) are also equivalent.

```
i1(X) :- t(X0), f(X0, X).                           % From F1
i1(X) :- i(X1), not p(X1), g(X1, X).                % From F2
```

Thus, with the following assumptions

1. the invariant of the imperative program and the type declaration of the Prolog program only refer to the "current" value of variables,
2. the loop/recursion always terminates successfully, and
3. the auxillary functions/procedures `t`, `f`, `g` and `h` have no preconditions/type declarations

then verifying that the Prolog program is type correct is equivalent to verifying the loop invariant of the imperative program using inductive assertions. In fact, the third assumption is not necessary; it was made to simplify the presentation. In general, additional implications need to be proved to show that calls are well typed or satisfy preconditions.

Let us now summarize how programs with type declarations can be verified. It requires two steps. One step is to show that the program is type correct, implying that any well typed answer computed by P is correct with respect to S. This step requires forward reasoning. The other step is to use consequence verification of S, to make sure that S is really what is intended. This requires backward reasoning, and can make use of the "type" information.

The analogous process for imperative programs is as follows. We first show, using inductive assertions, that some invariant `i1(X1)` holds. We then use subgoal induction to prove that the program computes the desired relation. In this second proof we can use the fact that `i1(X1)` is an invariant. This is new way of combining subgoal induction and inductive assertions, neither of which is best in all situations. In the gcd example, the invariant must include the fact that `x` and `y` are greater than zero. Because this is not implied by the exit condition of the loop (M1), subgoal induction cannot be used directly. We could use inductive assertions for the whole proof, but this requires a slightly more complicated loop invariant than subgoal induction. A compromise is to use inductive assertions to show that the variables are positive (this can be done if an extra test is added to the start of the code) and then use subgoal induction to show that the code computes the gcd function, given that the variables are positive.

10 Forward reasoning in Prolog programming

The fact that consequence verification does not seem adequate for all typical logic programs suggests that backwards reasoning alone is inadequate. The gcd programs and their associated correctness proofs we have presented also support this view. The best program, `gcd_r/gcd`, is most easily developed and explained using some forward reasoning at least. It is interesting to see the results of attempting to use forward reasoning exclusively in Prolog programming. Because the theory of types we have discussed allows very general definitions of "types", we can use types to define arbitrary invariants. It is then possible to reason forwards and use the loop invariants that would be used for inductive assertion verification of imperative programs, but code the result in Prolog with types declarations.

If we do this for the gcd program (not forgetting that the variables should be positive integers), the invariant is:

$$\mathtt{pos\_int(x)} \wedge \mathtt{pos\_int(y)} \wedge \mathtt{gcd(x,y)} = \mathtt{gcd(x0,y0)}$$

The code should establish this invariant by testing the inputs then loop, maintaining the invariant but not actually testing anything. The invariant becomes the type for a recursive procedure which has as arguments all the variables in the invariant, plus an extra output argument. We code the last constraint in the invariant ($\mathtt{gcd(x,y)} = \mathtt{gcd(x0,y0)}$) symbolically. A Prolog procedure could be written to test it, or it can just be used as an aid to documentation and verification.

```
gcd_f(X0, Y0, Z) :- pos_int(X0), pos_int(Y0), gcd_e(X0, Y0, X0, Y0, Z).

?- gcd_e(X, Y, X0, Y0, Z) type pos_int(X), pos_int(Y),
                 eq(gcd(X,Y), gcd(X0,Y0)).
gcd_e(Z, Z, X0, Y0, Z).
gcd_e(X, Y, X0, Y0, Z) :- X > Y, X1 is X - Y, gcd_e(X1, Y, X0, Y0, Z).
```

```
gcd_e(X, Y, X0, Y0, Z) :- X < Y, Y1 is Y - X, gcd_e(X, Y1, X0, Y0, Z).
```

Although the invariant uses the initial values of the variables, proving type correctness is still equivalent to using inductive assertions. We can think of the program as a translation of a Pascal program which copies the initial values of `x` and `y` into two extra variables. Proving that `gcd_f` is correct using consequence verification is trivial due to the type declaration. In fact, we don't even have to consider the clauses for `gcd_e`. Thus the verification only uses forward reasoning.

The code is correct and robust, and compiles into code which is almost as efficient as previous versions. However, the third and fourth arguments of `gcd_e`, `X0` and `Y0`, are only used in the type declaration; never in the clauses which are executed. They correspond to shadow variables in inductive assertion verification. Their purpose is to find `gcd(X0,Y0)`, which is the same as the last argument. If we eliminate the two redundant arguments, we get the same code as before. Thus, the advantage of reasoning backwards seems to be that simpler loop invariants are often produced. The difference is more acute in Prolog than in imperative languages since the relational notation used in Prolog means that the output variables (final values) must be passed around explicitly, so there is essentially no cost in using them in loop invariants.

Forward reasoning can lead to advantages other than more intuitive declarative semantics. It can lead to more efficient algorithms, by the use of accumulators. Consider the task of summing the elements in a list. The most concise definition, and the most obvious solution using backward reasoning is as follows:

```
sum([], 0).
sum(H.T, S) :- sum(T, S1), plus(S1, H, S).
```

When used as a Prolog program, this procedure requires O(N) stack space, where N is the length of the list to be summed. The numbers are actually summed from right to left.

The most obvious solution using forward reasoning is to sum the elements from left to right, maintaining an accumulator which is the sum of the elements encountered so far. The accumulator is initialized to zero. This leads to the following definition. Again, we code the invariant symbolically, noting that the sum of the elements so far encountered is the sum of the original list minus the sum of the current tail.

```
sum(L0, S) :- sum1(L0, L0, 0, S).

?- sum1(L, L0, A, S) type eq(A, sum(L0)-sum(L)).
sum1([], L0, S, S).
sum1(H.T, L0, A, S) :- plus(A, H, A1), sum1(T, L0, A1, S).
```

This program is tail recursive and can be run in a constant ammount of space. Note that again we have an argument (the second one) which is only used in the type declaration. This could be removed, though it would make the forward reasoning less clear and verification more difficult. Alternatively, we can ignore the type declaration and discover what the procedure actually computes. In this case it is quite simple to understand — `sum1(L,A,S)` is true if `S` is the sum of `L` plus `A`. We can therefore use this as the declarative semantics for the procedure and consequence verification can be used without verifying type correctness. This preferable coding could also be derived using backward reasoning, but requires more intuition than the original coding. We believe that even amongst those who espouse declarative Prolog programming, the declarative semantics of programs is often only understood in retrospect, after the program has been written.

We now consider a predicate which has always been defined using (mostly) backward reasoning — `append`. The normal definition relies on simple forward reasoning to maintain the invariant that all arguments are lists. Without this, an extra test would be needed in the base case to check that the arguments are lists. However, the rest of the reasoning is backwards. The invariant is that the final value of the third argument is the concatenation of the first two arguments, which is just a restatement of the `append` definition. Relying on the final value of the output variable in the invariant corresponds to backward reasoning: either consequence verification or subgoal induction.

To obtain a definition of `append` using only forward reasoning, it is helpful to consider how it can be coded using a while loop in an imperative language, and how this can be verified using inductive assertions. To construct the output list three pointers are needed. One points to the start of the list, one points to the end of the list and is moved along the list as it is constructed, and one points to and moves along the first input list. The invariant maintained at each stage is that the list constructed so far is the same as the part of the first input list which has been passed over. We can think of these two list segments being updated by moving the pointers and adding cells to the tail of the second list. The loop terminates when the end of the input list is reached. The tail of the other list can then be made to point to the second input list, then the head of the list can be returned as the final result.

The Prolog procedure corresponding to this reasoning has six arguments: the original first input list, the current tail of the list (corresponding to the pointer moving along the list), the second input list, the head and tail of the list segment being constructed, and the output list which is instantiated to the final value of the list segment which is built up. The list segments mentioned above correspond to difference lists in Prolog. The invariant, therefore, is that two difference lists are equal. This can be made explicit using a type declaration.

```
append(As0, Bs, Cs) :-
        app1(As0, As0, Bs, Ds, Ds, Cs).  % both d-lists empty initially

        % Args 1-2 & 4-5 are difference lists.  1-2 is the input list
        % processed so far, 4-5 is constructed list processed so far.
?- app1(A,AT, B, D,DT, C) type eq(D-DT, A-AT).
app1(As0, [], Bs, Cs, Bs, Cs).  % Ds = Bs, Cs = Ds0.
app1(As0, A.As, Bs, Ds0, A.Ds, Cs) :- app1(As0, As, Bs, Ds0, Ds, Cs).
```

Like the previous definitions using only forward reasoning, this definition has an argument which is never used. However, the basic operations needed to perform each recursive call are the same as those in the standard definition. There is a significant difference between this and the standard definition however. This definition loops when run backwards. Definitions constructed by only forward reasoning use accumulators and only instantiate the output arguments in the base case (which corresponds to the exit of a while loop). Instantiation of the output arguments cannot affect the recursive calls and prevent loops. This is a disadvantage of forward reasoning. In contrast, backward reasoning often results in reversible procedures.

11 Bottom-up logic programming revisited

As discussed earlier, consequence verification of bottom-up logic programs is equivalent to using inductive assertions, and requires forward reasoning. Nothing radical like a type system is needed to incorporate forward reasoning into bottom-up programs. However, there is a remarkable relationship between bottom-up logic programs and top-down logic programs which are written using only forward reasoning.

The Prolog programs written using forward reasoning presented in the last section all have a similar structure, corresponding to a while loop in an imperative program. The top level call sets up an invariant, the head of the recursive clause(s) take the values of variables on the Nth iteration of the loop, the recursive call gives the values on the N+1th iteration, and the base case implements the exit of the loop. Using bottom-up evaluation, the roles of the clause head and recursive call are reversed, as are the roles of the top level call and the base case. If variables are named consistently in the top level call and the base case, we can literally swap these two pairs of atoms to convert the top-down program into a bottom-up program. For example, the `sum` program is transformed as follows:

```
sum_b(L0, S) :- sum1_b([], L0, S, S).

sum1_b(L0, L0, 0, S).
sum1_b(T, L0, A1, S) :- plus(A, H, A1), sum1_b(H.T, L0, A, S).
```

Given a query such as `?- sum_b([1,2,3],X)`, the form of constrained bottom-up evaluation mentioned previously can be used. The second argument of `sum1` is not altered in the recursive call, so this argument of the top level call can be used in the base case. This results in the fact `sum1_b([1,2,3],[1,2,3],0,S)` being deduced. Using the recursive clause for `sum1`, we deduce `sum1_b([2,3],[1,2,3],1,S)`, then `sum1_b([3],[1,2,3],3,S)`, and finally `sum1_b([],[1,2,3],6,S)`. Using the clause for `sum_b`, we arrive at the answer, `X=6`. The fourth argument, which returned the answer in the top-down program, is now redundant. The second argument, which was redundant in the top-down program, now plays the role of constraining the base case.

The use of the second argument is also related to an extension of the NRSU transformation. The NRSU transformation [NRSU89] would allow the sum of a single list to be evaluated bottom-up without the second argument. The context transformation [KRS90] allows the sum of several lists to be computed at the same time by starting with several instances of the base case. An extra "context" argument is used to keep track of which list each intermediate result relates to. For example, if the lists `[1,2,3]` and `[5,2,3]` were summed, intermediate results would associate `[3]` to the partial sums 3 and 7. The context argument makes the association between these partial sums and the original lists. The second argument of sum1_b is the same as a context argument. The context transformation is essentially the same as the transformation we have described, except that it does not rely on the redundant argument in the original program, and it removes the redundant argument in the resulting program. Thus the relationship with the reasoning which can be used to construct the program is less clear.

It is not immediately obvious what relation sum1_b actually defines, given the unusual way the program was derived. From the relationship with the context transformation, we know that it defines the "magic predicate" for sum1 (see [BR87], for example), with additional arguments. It also defines the loop invariant/type we used for sum1: the third argument is the difference between the sums of the first two arguments, and the fourth argument is unconstrained. There seem to be strong links between our work and previous work on bottom-up evaluation. We are continuing to explore them.

Just as the use of the second argument of sum1 was made clear by a type declaration, so can the use of the fourth argument of sum1_b. And, just as the type of sum1 can be sum1_b, the type of sum1_b can be sum1 (the meaning without a type declaration). That is, the last argument is the sum of the elements in the first argument plus the third argument:

```
?- sum1_b(L, L0, A, S) type sum1(L, L0, A, S).
```

As we noted previously, the type scheme does not rely on any particular execution mechanism. With this type declaration in the program, consequence verification is made trivial. Showing that the program is type correct requires backwards reasoning; the same backward reasoning required to derive or verify sum1 without a type declaration. Thus, it is possible to use backward reasoning to arrive at an algorithm and code the result as a bottom-up logic program using type declarations to document the relationship between the "current" and "final" values of variables. Showing that a bottom-up logic program is type correct is equivalent to using subgoal induction for imperative programs.

12 Redundant type information

So far we have not considered "output" variables appearing in type declarations for top-down programs. This corresponds to using the final value of a variable in the invariant of an imperative program. If a procedure works in a particular input/output mode then the use of both input and output variables in type declarations is generally redundant. However, it can be more natural, especially when several modes can be used. Redundant information is useful if the type declarations are being used for detecting errors but can make proving type correctness more difficult in some cases. Consider the gcd_r program with the following type declaration:

```
?- gcd(X,Y,Z) type pos_int(X), pos_int(Y), pos_int(Z).
```

This is a natural way of expressing the intended types for the gcd procedure and the testing of Z could be used to detect extra errors in a system which executed type checks at runtime. However, checking that `gcd_r` is type correct now requires an analysis of the execution of `gcd`. We must show that when the first two arguments are positive integers and `gcd` succeeds, the third argument must also be a positive integer. In general, this kind of analysis is more complex than the analysis required when using inductive assertions for imperative programs. However, for right recursive procedures, such as `gcd` and those constructed using only forward reasoning, the output argument is not affected in the recursive clauses. It is then only necessary to analyse the base case.

Exactly what information is put into type declarations is a compromise. Type declarations can be used in many ways. Depending on how they are used, it may be best to avoid any redundancy, to express the programmers intentions most clearly, to include all invariants which are useful for verifying the program, or to include even more redundant information. Given this conflict, it may be best to develop some mechanism other than type declarations to allow the programmer to give additional information about the properties and intended semantics of procedures.

13 A larger example

We now discuss the various points we have made concerning verification of Prolog programs using a slightly larger example, a simple implementation of quicksort.

```
        % Ss is sorted version of Us, a list of numbers.
qsort(Us, Ss) :- qs(Us, [], Ss).

        % last two args form a 'difference list'
qs([], Ss, Ss).
qs(U.Us, Ss0, Ss) :-
        partition(U, Us, LEs, GTs),
        qs(GTs, Ss0, Ss1), qs(LEs, U.Ss1, Ss).

        % splits list into lists of elements =< and > key
partition(_, [], [], []).
partition(K, U.Us, U.LEs, GTs) :- U =< K, partition(K, Us, LEs, GTs).
partition(K, U.Us, LEs, U.GTs) :- U > K, partition(K, Us, LEs, GTs).
```

We want to show that `qsort` is true if and only if `Ss` is sorted and has the same multiset of elements as `Us`. We can write this condition in logic and try to prove it as a theorem, using the logic of the program and suitable definitions of sorted etc. However, this is difficult to do unless we structure the proof in some way. Without knowing the ideas behind quicksort, it is difficult to verify the program. Ideally, we should structure the proof in a similar way to the program, using the same ideas the programmer used to construct the program. For example, we could try to prove that `qs` preserves list elements and implies that `Ss` is sorted. For brevity, we just discuss sortedness. Unfortunately, `qs` can succeed with the third argument not sorted (it may not even be a list), despite the way the programmer may think of it.

Lets us consider the likely intended semantics for `qs`. We claim the programmer is unlikely to think of the result of calling `qs` with non-lists. The intended semantics is that all arguments are lists of numbers. If this was the only restriction, we could add the appropriate type declaration and the type correctness proof would only need reasoning about lists. This can be automated using the algorithms of [DZ92]. Consequence verification would need to reason about sortedness etc. However, if the programmer uses more forward reasoning, the intended semantics may have extra restrictions as well. The way `qs` is used, the second argument is a sorted list of numbers which are all greater than or equal to all the numbers in the first argument. The third argument is the second argument appended onto a sorted version of the first argument (hence it must be sorted). These extra restrictions on the second argument can be made explicit with the following type declaration:

```
?- qs(A, B, C) type
        sorted(B), all_le(A, B).
```

There are several ways in which we can think of this declaration. First, it is a way the programmer can document the intended semantics of the predicate. Second, it provides some additional information to help structure the verification proof of the program. Third, it is an important "loop" invariant in this quicksort program. Given these intended semantics, it is trivial to prove that (in the intended semantics) the second argument of `qsort` is indeed sorted.

Proving that `qs` is type correct is more difficult than previous type correctness proofs. We must show that the elements of the second arguments of the recursive `qs` calls are greater than or equal to the elements of the first arguments. This requires reasoning about the answers returned by `partition`, rather than analysis of types. This can be explained by noting that the program uses backward reasoning, incrementally constructing the output arguments of `partition` in a similar way to the normal version of `append`.

However, `partition` can be rewritten in a way which uses just forward reasoning. One method is to use difference lists, as we did with `append`. The program below uses a slightly simpler method. Two accumulators are used to maintain the current lists of elements greater than and less than the pivot element. The accumulators are always ground, and the more complicated notion of difference lists is not needed. One consequence is that the output lists are reversed. However, this does not affect the correctness of quicksort. We document some of the forward reasoning using a type declaration. To make all the forward reasoning explicit, a redundant argument, the original list, should be passed to `pt`, and the type declaration should relate the elements of the accumulators to this.

```
partition(K, Us, LEs, GTs) :- pt(K, Us, [], LEs, [], GTs).

        % args 3 and 5 are accumulators
?- pt(K, Us, LEs0, LEs, GTs0, GTs) type
                number(K), le(LEs0, K), gt(GTs0, K).
pt(_, [], LEs, LEs, GTs, GTs).
pt(K, U.Us, LEs0, LEs, GTs0, GTs) :-
        U =< K, pt(K, Us, U.LEs0, LEs, GTs0, GTs).
pt(K, U.Us, LEs0, LEs, GTs0, GTs) :-
        U > K, pt(K, Us, LEs0, LEs, U.GTs0, GTs).
```

The proof of type correctness is now more straightforward, and very similar to an inductive assertion proof. Since the fourth and sixth arguments of `ps` are not effected in the recursive clauses, we only need to consider the base case when proving that they are well typed when the other arguments are well typed (like `gcd`).

Although the proof is very similar a technique for verifying sequential imperative programs, the program can still use alternative evaluation strategies, such as parallel execution of the two `qs` calls. As long as one possible evaluation corresponds to the execution of a simple imperative program, the type correctness proof can be similar to using inductive assertions. Whether the extra work verifying the previous version of the program is avoidable is not currently known. Incremental instantiation of terms is most easily handled by backwards reasoning, but there might also be techniques for easily proving type correctness of such programs.

To summarise, the verification of `qsort` changes depending on how much type information is given. If the programmer uses mostly forward reasoning (as a Pascal programmer generally would), detailed type information is given and the relations in the intended interpretation correspond very closely to what is computed by the procedures in practice. Consequence verification becomes very simple but proving type correctness becomes complex. If the programmer uses more backwards reasoning then the type information is less precise and the relations in the intended interpretation are larger than what is actually needed in practice, since the calls are restricted. Consequence verification is then a more substantial part of the proof, though the proof is simpler overall. It is possible that the programmer uses no forward reasoning at all. The type correctness proof then becomes trivial, but the relations in the intended interpretation become less intuitive. A more likely result of using only backward reasoning is a less efficient version of quicksort which does not use an accumulator.

14 Conclusion

Formal verification techniques can explicate the reasoning used to construct programs. We have examined forward reasoning, as exemplified by the use of the inductive assertion method for imperative programs and consequence verification for bottom-up logic programs, and backward reasoning, as exemplified by the use of subgoal induction for imperative programs and consequence verification for top-down logic programs. Backward reasoning alone does not result in efficient top-down logic programs (for example, Prolog programs) unless the intended declarative semantics are very unintuitive. However, forward reasoning can also be used to construct these programs.

The equivalent of loop invariants can be documented using "type" declarations. Verification can then be done in two parts: proving that the program is type correct and performing consequence verification of the program with additional type checks added. The forward reasoning component in the construction of the program corresponds to the proof of type correctness and the backward reasoning component corresponds to the consequence verification. Proving that a program is type correct is equivalent to using the inductive assertion method, so this verification method corresponds to a combination of the inductive assertion method and subgoal induction. An equivalent technique can be used for imperative programs.

Both forward and backward reasoning have advantages in Prolog. Backward reasoning tends to result in code with fewer arguments and procedures are often reversible. Forward reasoning can lead to type checks being avoided and accumulators being used efficiently without the declarative semantics of procedures being unintuitive. Generally, a mixture of forward and backward reasoning results in the best Prolog programs. If only forward reasoning is used, the code is efficient for top-down evaluation but has redundent arguments and is not reversible. However, the code is also amenable to efficient bottom-up evaluation, by using a simple program transformation. There are close links between our use of type declarations for forward reasoning and previous work on efficient bottom-up evaluation of logic programs. We are continuing to investigate this area.

Acknowledgements

I would like to thank Rodney Topor for help with searching the literature, and Justin Zobel for comments on an earlier draft.

References

[BR87] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[CD88] B. Courcelle and P. Derensart. Proofs of partial correctness for attribute grammars with application to recursive procedures and logic programming. *Information and Computation*, 78(1):1–55, July 1988.

[Cla84] K. Clark. The synthesis and verification of logic programs. In J.N. Crossley and J. Jaffar, editors, *Proceedings of the Logic and Computation Conference*, Melbourne, Australia, January 1984.

[CS77] K. Clark and S. Sickel. Predicate logic: a calculus for the formal derivation of programs. In *Proceedings of the Fifth IJCAI*, pages 419–420, Cambridge, Massachusetts, August 1977.

[CvE81] K.L. Clark and M.H. van Emden. Consequence verification of flowcharts. *IEEE Transactions on Software Engineering*, SE-7(1):52–59, January 1981.

[Dij79] E.W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[DM88] Wlodzimierz Drabent and Jan Maluszynski. Inductive assertions method for logic programs. *Theoretical Computer Science*, 59:133–155, 1988.

[DZ92] Philip W. Dart and Justin Zobel. Efficient runtime type checking of typed logic programs. To appear, *Journal of Logic Programming*, 1992. Revised version of technical report 88/11, Department of Computer Science, University of Melbourne, Australia.

[Flo77] R.W. Floyd. Assigning meanings to programs. In T.J. Schwartz, editor, *Proc. Symp. Appl. Maths*, volume XIX, pages 19–32, Providence, 1977. RI: Amer. Math. Soc.

[Gri81] D. Gries. *The science of programming.* Texts and Monographs in computer science. Springer-Verlag, New York, 1981.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576, October 1969.

[Hog77] C. Hogger. Deductive synthesis of logic programs. Technical report, Department of computing, Imperial College, London, 1977.

[Kow80] R. Kowalski. *Logic for problem solving.* North Holland, New York, 1980.

[Kow85] R. Kowalski. The relation between logic programming and logic specification. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical logic and programming languages*, Series in Computer Science, pages 11–27. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[KRS90] David B. Kemp, Kotagiri Ramamohanarao, and Zoltan Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 380–391, Brisbane, Australia, August 1990.

[MW77] J.H.Jr. Morris and B. Wegbreit. Subgoal induction. *Communications of the ACM*, 20(4):209–222, April 1977.

[Nai90] L. Naish. Types and the intended meaning of logic programs. Technical Report 90/4, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1990. To appear in *Types in Logic Programming*, MIT Press.

[NRSU89] J.F. Naughton, R. Ramakrishnan, Y. Sagiv, and J.D. Ullman. Efficient evaluation of right-, left- and multi-linear rules. In *Proceedings of the ACM SIGMOD '89*, pages 235–242, 1989.

[Top77] R. Topor. Some new techniques for verifying programs. In R.B. Stanton M.C. Newey and G.L. Wolfendale, editors, *Proceedings of the Conference on Programming Language Systems*, pages 191–202, Canberra, Australia, February 1977. Australian National University Press.