PROVING ASSERTIONS IS ALSO USEFUL

JÜRGEN F. H. WINKLER, STEFAN KAUER

Friedrich Schiller University, Institute of Computer Science, D-07740 Jena, Germany
http://www1.informatik.uni-jena.de
Appeared in: SIGPLAN Notices, 32,3 (1997) 38 .. 41

"Mathematics and Computer Science: Coping with Finiteness"
D.E. Knuth, 1976

Proving Assertions

Mike Marin's paper [Mar 96] is a good example for the use and benefits of assertions in program development. Assertions can be applied in two ways:

a) as dynamic checks: the condition of the assertion is evaluated during program execution. If the value of the condition is false an error is reported. This is then a hint to the programmer to improve the program. This possibility is the topic of [Mar 96] (see e.g. also [Luc 90; Mag 93]). If the value of the condition is true this shows that the assertion is fulfilled for the current set of input data, but only for this specific set.

b) as a condition which must always hold when program execution reaches the point of the assertion. In some cases this can be proved before the program is executed. This possibility is the topic of the present paper. Such a proof means that the assertion is fulfilled for all possible sets of input data.

If we compare both approaches we see that b) is more general, but on the other hand, it is usually also more difficult than a). Both are useful in that the formulation of (sharp) assertions usually improves the understanding of the program and the underlying algorithm.

In order to be able to prove properties of programs the semantics of programs and their components has to be defined in an exact manner. Instead of semantics we may also speak of the "effect" of a program or a program fragment. Examples for such definitions are Hoare rules [Hoa 69] or the weakest precondition (wp) of Dijkstra [Dij 76; Gri 83].

A simple example is the definition of the effect of the simple assignment statement:

```
V := Expr;
```

We assume that V is a simple variable and that the evaluation of Expr does not cause any side effects.

In the wp-calculus the effect of this statement is given by:

$$wp(\text{"V := Expr;"}, Post) \equiv type_V(Expr) \wedge Post^V_{Expr}$$

where $type_V(Expr)$ means that the value of Expr is one of the values of the type of V and $Post^V_{Expr}$ means that the variable V is substituted in Post by the expression Expr. Informally $Post^V_{Expr}$ says: whatever holds of Expr immediately before "V := Expr;" is executed will hold immediately after the execution of the assignment. A simple example for this is (let v be of type int):

```
v := v + 1;
   --!Post:  v > 0;
```

For this we have

$$\begin{aligned} & wp(\text{"v:=v+1;"}, v>0) \\ \equiv\ & -2^{**}31 <= v+1 \wedge v+1 <= 2^{**}31 -1 \wedge (v>0)^v_{v+1} \\ \equiv\ & -2^{**}31 <= v+1 \wedge v+1 <= 2^{**}31 -1 \wedge v+1 > 0 \\ \equiv\ & 0 <= v \wedge v <= 2^{**}31 -2 \end{aligned}$$

If the assignment is preceded by the initialization:

```
v := 1;
```

we obtain

```
v :=1;
   --!Post: 0 <= v   ∧   v <= 2**31 -2;
   --!Pre:  0 <= v   ∧   v <= 2**31 -2;
v := v+1;
   --!Post: v > 0;
```

If we apply again the wp-rule for the initialization we obtain:

wp("v:=1;", 0 <= v ∧ v <= 2**31 -2)
≡ (0 <= v ∧ v <= 2**31 -2)$^{v}_{1}$
≡ 0 <= 1 ∧ 1 <= 2**31 -2
≡ true

Since "true" is always true the result of our calculation is: we may start "v:=1; v:=v+1;" in any program state and have the guarantee that immediately after the execution the assertion "v > 0" will hold.

On the other hand, if we have

```
v := 11;
   --!Post: v=1;
```

we obtain:

wp("v:=11;", v=1)
≡ -2**31 <= 11 ∧ 11 <= 2**31 -1 ∧ (v=1)$^{v}_{11}$
≡ 11 = 1
≡ false.

Since "false" is never true there exists no state in which "v:=11;" may be started if its execution should establish "v=1". So we have a strong hint that the program or the assertion is not correct (in this case this may just be due to a typo).

As the examples show the calculation of such weakest (i.e. most general) preconditions is in some cases a mechanical process which can be done by the computer.

A tool which supports such calculations is developed in our group. The tool is called the Frege Program Prover (FPP) and can be used via WWW under (http://www1.informatik.uni-jena.de/FPP/FPP-main.htm).

FPP can also prove that a triple

{Pre} S {Post}

is consistent, i.e. that the following implication does hold in all program states:

Pre ⇒ wp(S,Post)

Such a proof is often more difficult than computing weakest preconditions for assignment statements. One consequence is that FPP may not be able to find a proof for such an implication. One such example is:

{true} NULL; {Goldbach's Conjecture}

Factorial Revisited

The use of assertions is shown in [Mar 96] using the factorial function (p.30). Marin uses an iterative implementation of factorial which can in this form be found in textbooks (in most cases without the invariant). Unfortunately, this implementation of the factorial function is not completely correct.

There are two defects in this program:

a) The loop invariant "r >= (i-1)" is not strong enough to guarantee the postcondition "r = n!". Such a situation does not automatically mean that the program is not correct, because a program which fulfills an assertion A1 usually also fulfills an assertion A2 which is weaker than A1. This is also the case in this example. The basic idea of the program is correct, but due to the second defect (see b)) the program is correct only for a tiny fraction of the possible input data values.

b) The program implements the factorial function as a mapping: int → long. If we assume that int means 32-bit numbers and long means 64-bit numbers (as e.g. in Java [GJS 96: 31]) any value in -2**31 .. 2**31-1 is a legal input. If we take e.g. the value 1000 we see easily that 1000!, which has 2568 decimal digits, is not in the range of long, because the largest number in long has only 19 decimal digits.

Both defects are the reason that FPP says, that it cannot prove that the given loop computes the factorial of n:

```
   --!pre : r = 1;
   --!post: r = Factorial(n);
   --!inv : r >= i-1;
for i in 1 .. n  loop
   r := r * i;
end loop;
```

Result:

```
--!pre       : r = 1
--!post      : r = Factorial(n)
--!inv       : r >= -1 + i
-->functionality --------------------
-->func: (initial AND induction AND
-->        final AND null loop)
-->initial:1 <= n AND r = 1 ==> r >= -1
-->Result    : proved
-->induction :1 <= n AND r >= -2 + i
-->        ==> i*r >= -1 + i
-->Result    : not proved
-->final     : 1 <= n AND r >= -1 + n
-->         ==> r = Factorial(n)
-->Result    : not proved
-->null loop :1 >= 1 + n AND r = 1
-->        ==> r = Factorial(n)
-->Result    : not proved
```

```
for i in 1 .. n  loop
   r := r * i;
end loop;
```

The syntax used in this example is Ada, because FPP supports currently a subset of Ada.

If we look at both defects we obtain:

a) an invariant, which is sufficient to derive the postcondition, is "r = (i-1)!"

b) A closer analysis of the factorial function [NW 89] reveals that only values up to 20 lead to a correct result:

   20! = 2_432_902_008_176_640_000
   2**63-1 = 9_223_372_036_854_775_807
   21! = 51_090_942_171_709_440_000

   This means that factorial works correctly for 21 out of 2_147_483_648 possible input values (we ignore the negative values for which the program always returns 1); the admissible input values are $9.77889\ 10^{-7}$ % of the possible input values. We may therefore say that the program is incorrect for almost all possible inputs.

   Since C compilers usually do not generate code to react to arithmetic overflow the given program is especially fallacious. If we compute 23! we obtain 8128291617894825984 which is quite different from the correct result. (The computation was done with gcc on OSF1 V3.2). In Java it seems not even to be allowed for the compiler to generate such code [GJS 96: 351, 352]. In such a case a thorough analysis of the algorithm before implementation is even more important.

   For $n < 0$ the program always yields r=1. Whether this is wrong or not depends on the definition of the factorial function. In [Knu 69: 45] for example factorial is not defined for $n < 0$, and Mathematica [Wol 94: 49] gives the definition "product of the integers 1, 2, ..., n". Therefore, we formulate the improved program for the factorial function for the argument range $0 \le n \le 20$.

With these observations in mind, we obtain an improved loop for the computation of the factorial function:

```
--!pre : r = 1 AND 0 <= n AND n <= 20;
--!post: r = Factorial(n) AND
--!post: -2**63 <= r AND r <= 2**63-1;
--!inv : r = Factorial(i) AND
--!inv : 0 <= n AND n <= 20;
for i in 1 .. n  loop
   r := r * i;
end loop;
```

The invariant "r = i!" is different from that mentioned above ("r = (i-1)!"). This is due to the fact that FPP uses a scheme based on [Hoa 72] to prove the correctness of FOR-loops.

The answer of FPP to this improved loop is:

```
--!pre : r = 1 AND n >= 0 AND n <= 20
--!post: r = Factorial(n) AND
--!post: -9223372036854775808 <= r AND
--!post: r <= 9223372036854775807
--!inv : r = Factorial(i) AND n >= 0
--!inv : AND n <= 20
-->functionality ----------------------
-->func: (initial AND induction AND
-->        final AND null loop)
-->initial: 1 <= n AND r = 1 AND
-->          n >= 0 AND n <= 20
-->     ==> r = 1 AND n >= 0 AND n <= 20
--> Result   : proved
--> induction: 1 <= n AND
-->            r = Factorial(-1 + i)
-->            AND n >= 0 AND n <= 20
-->        ==> i*r = Factorial(i) AND
-->            n >= 0 AND n <= 20
--> Result: proved
--> final : 1 <= n AND r = Factorial(n)
-->         AND n >= 0 AND n <= 20
-->     ==> r = Factorial(n) AND
-->     -9223372036854775808 <= r
-->     AND r <= 9223372036854775807
--> Result: proved
--> null loop: 1 >= 1 + n AND r = 1 AND
-->            n >= 0 AND n <= 20
-->        ==> r = Factorial(n) AND
-->           -9223372036854775808 <= r
-->         AND r <= 9223372036854775807
--> Result    : proved
for i in 1 .. n  loop
   r := r * i;
end loop;
```

All four parts of the functionality condition have been proved and therefore the system loop+assertions is consistent. Since the postcondition states the desired result we have a correct implementation of the factorial function.

A more *efficient* implementation of the factorial function can be found in [NW 89].

Conclusion

As a conclusion we would like to appeal to

a) the computing community at large:

to accept wholeheartedly and from the very beginning the fact that computers are finite machines and that therefore we have only finite ranges for numbers. If this is accepted one consequence is, that mathematical formulae cannot always be implemented by just transliterating them into a programming notation (e.g. “mean := (a+b)/2.0;” works for less values of a and b than “mean := a/2.0 + b/2.0;”).

b) to the processor industry:

it would be very useful to define an integer arithmetic which takes the finite ranges fully into account. It would especially be useful to have two reserved bit patterns for -∞ and for +∞. For floating point there is e.g. IEEE 754, but for integers there is no such definition. (ISO/IEC 10967-1:1994(E) is quite conservative.)

If the details are not correct,
how should the big system be correct ?

References

Dij 76 Dijkstra, Edsger W.: A Discipline of Programming. Prentice-Hall, 1976. 0-13-215871-X

GJS 96 Gosling, James; Joy, Bill; Steele, Guy: The Java™ Language Specification. Addison-Wesley 1996. 0-201-63451-1

Gri 83 Gries, David: The Science of Programming. Springer, 2nd pr. 1983. 0-387-90641-X

Hoa 69 Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. CACM 12,10 (1969) 576..580, 583

Hoa 72 Hoare, C.A.R.: A Note on the FOR Statement. BIT 12,3 (1972) 334..341

Knu 69 Knuth, Donald E.: The Art of Computer Programming. Vol.1. 2nd pr. Addison-Wesley 1969

Knu 76 Knuth, Donald E.: Mathematics and Computer Science: Coping with Finiteness. Science 194, 4271 (1976) 1235..1242

Luc 90 Luckham, David: Programming with Specifications. Springer 1990. 0-387-97254-4

Mag 93 Maguire, Steve: Writing Solid Code. Microsoft Press, Redmond, 1993. 1-55615-551-4

Mar 96 Marin, Mike A.: Effective use of Assertions in C++. SIGPLAN Not. 31,11 (1996) 28..32

NW 89 Nievergelt, J.; Winkler, J.F.H.: How should the factorial function be computed ?. Informatik Spektrum 12,4 (1989) 220..221 (in German)

Wol 94 Wolfram, Stephen: Mathematica. 2. Aufl. Addison-Wesley 1994

1997.Feb.09
1997.May.24