JÜRGEN F. H. WINKLER

THE FREGE PROGRAM PROVER FPP

Abstract

The Frege Program Prover (FPP) is an experimental system which supports the programmer when calculating the effect or semantics of programs or program fragments. FPP supports two kinds of such calculations:

a) *Compute the weakest precondition*: wp(PF, Post). Compute the weakest precondition for a given program (fragment) PF and a given postcondition Post.
   Example: wp(„v:=v+1;“, v>4)
   ≡ v+1 ∈ type(v) ∧ v+1 > 4
   ≡ v+1 ∈ type(v) ∧ v ≥ 4.

b) *Check the correctness of a program* (fragment): i.e. check whether a given program (fragment) PF satisfies a given specification (Pre,Post). This is usually expressed as a Hoare triple {Pre} PF {Post}. If PF satisfies the specification the triple is called consistent. This consistency is defined as: Pre ⇒ wp(PF,Post).
   Example: {v>0} v:=v+1; {v>4}
   ≡ v>0 ⇒ wp(„v:=v+1;“, v>4)
   ≡ v>0 ⇒ v+1 ∈ type(v) ∧ v ≥ 4
   ≡ False.

FPP is implemented as a WWW application (http://www1.informatik.uni-jena.de/FPP/FPP-main.htm) and supports a subset of Ada: Integer and Boolean variables, assignment, sequence, IF, CASE, FOR, and WHILE.

1 Introduction

Software has become an important and often essential part of technical systems. It is used in small devices as e.g. pacemakers up to big systems as e.g. airplanes. Both are examples of systems which require a high degree of reliability and safety. This means that all parts of design must be checked whether they guarantee these requirements. For electrical or mechanical designs there exist methods to check properties in an exact manner: e.g. circuit analysis [Dor 93] or analysis of structure. Such methods are routinely applied by electrical or civil engineers.

In SW engineering such methods also exist but are not as mature as those in other fields of engineering and are therefore neither taught nor used routinely during the development of programs. Whereas any electrical engineer can perform the calculations necessary for the quantitative analysis of simple circuits it is usually not he case that a software engineer can calculate the effect of a simple program or program fragment in a quantitative way. One important reason for this is the very rapid development of the field.

The Frege Program Prover (FPP) is an experimental system to support the SW engineer in such calculations.

FPP is implemented as a WWW application (http://www1.informatik.uni-jena.de/FPP /FPP-main.htm) and supports a subset of Ada: Integer and Boolean variables, assignment, sequence, IF, CASE, FOR, and WHILE.

2 Semantics or the Effect of Programs

As Dijkstra has pointed out it is the task of computers to execute the programs constructed by the SW engineers. The purpose of program execution is to put the computer in some specific state. Such a required final state can e.g. be a state in which a certain variable contains the sum of some other variables. In programs running continuously, as e.g. in embedded systems, this characterization holds for each cycle. The effect of a program can therefore be characterized by the relation between the initial state (in which it is started) and the final state(s) (in which it terminates). This is often also called the semantics of the program. In this paper I use mostly the term „effect“.

The most abstract description of the effect of a program is the relation between initial and final states [Win 96]. Usually programs are constructed out of smaller pieces (as. e.g. modules, routines, declarations, statements).

In order to compute the effect of the whole program two things must be known: (1) the effect of the smaller pieces and (2) how to compute the effect of a composition. Example: if we know the effect of the two assignments „abs := x;“ and „abs := -x;“ we have to be able to compute the effect of:

```
IF  min < x < 0
THEN  abs := -x;
ELSE  abs := x;   FI
```

The program state is therefore the basic entity for the characterization of the effect of programs. The program state is characterized by the values of all objects in the program. Especially important among these objects are the variables. In this paper we will characterize the program state by the values of the variables in the program [Win 96] :

DEF-1: $V = \{ v_1, v_2, ..., v_n \}$ is the set of program variables, where $n \geq 1$.

$W_1, W_2, ..., W_n$ are nonempty, finite sets of admissible values for the corresponding v_i: $W_i = VS(v_i)$.

$W = W_1 \cup W_2 \cup ... \cup W_n$.

$S = \{s \in V \rightarrow W: (\forall v \in V: s(v) \in VS(v))\}$ is the set of proper states, where $V \rightarrow W$ denotes the set of functions from V into W.

$S_\perp = S \cup \{\perp\}$ is the set of states.

$C = 2^S$ is the set of proper conditions.

$C_\perp = 2^{S\perp}$ is the set of conditions.

□ DEF-1

We call subsets of S or $S_\perp$ conditions because these subsets can be characterized by logical expressions E where the logical expression acts as the characteristic predicate of the subset: e.g. $\{s \in S: E(s)\}$. Often it is simpler to characterize a set of states (e.g. those in which $x > 0$ for some program variable x) by a logical expression than by explicitly writing down the set of states. In the computation of the effects of programs we often deal with sets of states rather than with single states. Therefore conditions play a prominent role in program analysis. The term „program analysis“ is used in analogy to „circuit analysis“ [Dor 93, Ch. 3].

DEF-2: $PROG = \{P \subseteq S \times S_\perp: dom(P) = S \}$ is the set of abstract programs.

An element $(i,f) \in S \times S_\perp$ is called a computation with initial state i and final state f.

If $f \neq \perp$ then we say the computation (i,f) terminates properly; if $f = \perp$ then the computation is a not(properly terminating) one. This behavior can occur in different forms: e.g. infinite loop or stop in the „middle“ of the computation ; therefore the somewhat unconventional expression „not(properly terminating)“. □ DEF-2

In order to see how these abstract definitions work for programs we look at some examples.

Exmp 1: the effect of the assignment x:= 10; can be characterized as follows:

$$\{ 10 \in VS(x) \} \quad x := 10; \quad \{ x = 10 \}.$$

This triple „condition statement condition“ is a proposition which states: whenever the initial state is such that „$10 \in VS(x)$“ holds then x:=10; is executable in this state and after termination the program is in a state in which x=10 holds. The precondition, in this case, is very simple: it is either true or false. If it is true it denotes the set S of all proper program states: $\{s \in S: true\}$. If it is false it denotes the empty set: $\{s \in S: false\}$. In the latter case there exists no program state in which x:=10; may be started and executed successfully.

Exmp 2: a second example is the assignment x:=x+1; . Its effect can be characterized by:

$$\{x=Kx \wedge x, x+1 \in VS(x) \}$$
$$x:=x+1;$$
$$\{x=Kx+1 \wedge x \in VS(x)\}$$

If we assume VS(x) = 1..100 we see more directly the sets of states characterized by the precondition and by the postcondition:

$$\{ x=Kx \wedge 1 \leq x, x+1 \leq 100 \}$$
$$x:=x+1;$$
$$\{ x=Kx+1 \wedge 1 \leq x \leq 100 \}$$

which may be transformed into:

$$\{ 1 \leq x \leq 99 \wedge x=Kx \}$$
$$x:=x+1;$$
$$\{ 2 \leq x \leq 100 \wedge x=Kx+1 \}$$

In this case there are several states, in which x:=x+1; may be successfully executed: $1 \le x \le 99$. If x=100 then an overflow will occur. The detailed effect of x:=x+1; is described using the specification variable Kx which expresses the relation between x before and after the execution of x:=x+1; .

Exmp 3: a third example is a conditional statement which computes for certain values of x the absolute value of x.
Let VS(x) = -128 .. +127.

{ x=-13 }
IF x < 0 THEN x := -x; FI
{ x = |-13| = 13 }

A more detailed annotation shows that the statement really computes the absolute value of the initial value of x:

{ x=-13 }
IF x < 0 THEN
{ x=-13 ∧ -128 ≤ -13 ≤ 127 }
x := -x;
{ -x=-13 ∧ -128 ≤ 13 ≤ 127 }
FI
{ x = |-13| = 13 }

Further simplification yields:

{ x=-13 }
IF x < 0 THEN
{ x=-13 } x := -x; { -x=-13 }
FI
{ x = |-13| = 13 }

In this case we have only looked at a single starting state for the program (x = -13). In section 4 we will do a more thorough analysis of this statement.

3 The wp-Calculus

In section 2 we have seen some examples of the precise characterization of the effect of statements. The treatment was somewhat ad hoc and we left it to the reader to be convinced of the correctness of the annotated program fragments. A more systematic treatment is possible if we define the relation between the precondition and the postcondition of a statement in a precise way. One possibility for this is the wp-calculus which computes for a given pair (statement, postcondition) the weakest i.e. most general precondition wp(statement, postcondition) such that (1) the statement is successfully executable in any state of the wp(statement, postcondition), and (2) the state after the execution of statement is in postcondition [Dij 76; DS 90]. If we express it in the language of states wp(statement, post) is the largest set of states such that (1) the statement is successfully executable in any state of wp(statement, post), and (2) the state after the execution of statement is in post [Win 96].

The effect of an assignment to a simple variable is defined by:

$$wp(\text{„x:=expr;“}, postcond) \equiv def(expr) \wedge postcond^{x}_{expr}$$

This means that (1) if expr is evaluated before the execution of the assignment it must be well defined and its value must be in VS(x), and (2) whatever holds for the value of x after the execution holds for the value of expr before the execution.

Exmp 4: (let VS(x) = 1..100)

$$wp(\text{„x:=x+2;“}, x > 50) \equiv$$

$$1 \le x+2 \le 100 \wedge [x > 50]^{x}_{x+2} \equiv$$

$$-1 \le x \le 98 \wedge x+2 > 50 \equiv$$

$$49 \le x \le 98$$

If the expression on the right hand side is more complicated the rule for def(expr) must also express appropriate constraints for the subexpressions.

The effect of an IF-statement is defined by:

$$wp(\text{„IF be THEN stat1 ELSE stat2 FI“}, poc) \equiv$$

$$def(be) \wedge [\, be \wedge wp(stat1, poc) \vee \neg be \wedge wp(stat2, poc) \,]$$

This rule expresses the following:

a) the expression be must be welldefined in the state before execution of the IF-statement

b) the evaluation of be does not change the program state

c) if the value of be is true then stat1 is executed

d) if the value of be is false then stat2 is executed

This is what we expect to hold for the IF-statement.

Exmp 5: we want to show that the following IF-statement computes the absolute value of the initial value of x (let VS(x) = -128 .. 127):

$$\{-128 \le x \le 127 \ \wedge \ x = Kx\}$$

IF x<0 THEN x:=-x; ELSE null; FI

$$\{ -128 \le x \le 127 \ \wedge \ x = |Kx| \}$$

wp(„IF x<0 THEN x:=-x; ELSE null; FI“, x=|Kx|) ≡

$$def(x<0) \ \wedge \ [x<0 \ \wedge \ wp(„x:=-x;“, x=|Kx|) \ \vee \ x \ge 0 \ \wedge \ wp(„null;“, x=|Kx|)] \ \equiv$$

$$-128 \le x \le 127 \ \wedge \ [x<0 \ \wedge \ -128 \le -x \le 127 \wedge -x=|Kx| \ \vee \ x \ge 0 \ \wedge \ x=|Kx|)] \ \equiv$$

$$-128 \le x \le 127 \ \wedge \ [-127 \le x \le -1 \wedge -x=|Kx| \ \vee \ 0 \le x \le 127 \wedge \ x=|Kx|)] \ \equiv$$

$$[-127 \le x \le -1 \wedge -x=|Kx| \ \vee \ 0 \le x \le 127 \wedge \ x=|Kx|)] \ \equiv$$

$$-127 \le x \ \le 127$$

We see immediately that the IF-statement does not work for x=-128 because the weakest precondition does not hold in this state:

$$-127 \le -128 \ \le 127 \ \equiv \ \text{false}$$

4 The Frege Program Prover

It is quite tedious and error prone to perform the calculations of the wp-calculus by hand. This is a typical task for the computer. The Frege Program Prover (FPP) is a tool to perform such calculations. FPP can essentially do two things:

a) *Compute the weakest precondition*: wp(PF, Post). Compute the weakest precondition for a given program (fragment) PF and a given postcondition Post.
Example: wp(„v:=v+1;“, v>4)
$\equiv \ v+1 \in VS(v) \ \wedge \ v+1 > 4$
$\equiv \ v+1 \in VS(v) \ \wedge \ v \ge 4.$

b) *Check the correctness of a program* (fragment): i.e. check whether a given program (fragment) PF satisfies a given specification (Pre,Post). This is usually expressed as a Hoare triple {Pre} PF {Post}. If PF satisfies the specification the triple is called consistent. This consistency is defined as: Pre $\Rightarrow$ wp(PF,Post).
Example: {v>0} v:=v+1; {v>4}
$\equiv \ v>0 \ \Rightarrow$ wp(„v:=v+1;“, v>4)
$\equiv \ v>0 \ \Rightarrow \ v+1 \in VS(v) \ \wedge \ v \ge 4$
≡ False.

FPP is implemented as a WWW application (http://www1.informatik.uni-jena.de/FPP/FPP-main.htm) and supports a subset of Ada: Integer and Boolean variables, assignment, sequence, IF, CASE, FOR, and WHILE.

If we apply the FPP to the examples given earlier in the paper we obtain:

Exmp 1: *Input to FPP:*

```
  --!Pre: -128 <= x and x < 127;
x := 10;
  --!Post: x = 10;
```

Answer of FPP:

```
1) --!pre:-128 <= x AND 127 >= 1
+ x
2) --> wp: True
3) --> vc: True
4) --> Result: proved
5) x := 10;
6) --!post: x = 10
```

The answer of FPP consists of the original input (lines 1, 5, and 6) and the result of the attempt to proof that the proposition

$$\{ -128 \le x \le 127 \} \quad x := 10; \quad \{ x = 10 \}$$

is true. In FPP this proposition is called the verification condition (vc). As mentioned above

$$vc \ \equiv \ -128 \le x \le 127 \Rightarrow \ wp(„x:=10;“, x=10)$$

FPP reports the weakest precondition in line 2 and the vc in line 3. In line 4 FPP says whether it could prove the vc. Since the vc is already true the proof is in this case a very simple one and has been successfully performed by FPP. That vc ≡ true can be seen as follows: wp(„x:=10;“, x=10) ≡ 10 = 10 ≡ true (FPP does in this step not take into account VS(x)). With this we obtain:

vc ≡ $-128 \le x \le 127 \Rightarrow$ true ≡ true.

Exmp 2: *Input to FPP:*

```
--!Pre: x=kx and 1 <= x and
--!Pre: x <= 100 and 1 <= x+1 and
--!Pre: x+1 <= 100;
x := x+1;
--!Post: x=kx+1 and 1 <= x
--!Post: and x <= 100;
```

Answer of FPP:

```
--!pre: x = kx AND 1 <= x AND
--!pre: x <= 100 AND 1 <= 1 + x
--!pre: AND 1 + x <= 100
--> wp: 1 + x = 1 + kx AND
-->     1 <= 1 + x AND
-->     1 + x <= 100
--> vc:   x = kx AND 1 <= x
-->   AND x <= 100 AND 1 <= 1 + x
-->   AND 1 + x <= 100
-->  ==>  1 + x = 1 + kx
-->   AND 1 <= 1 + x
-->   AND 1 + x <= 100
--> Result: proved
x := x + 1;
--!post: x = 1 + kx AND 1 <= x
--!post: AND x <= 100
```

The answer of FPP shows that this program fragment is consistent.

Exmp 3: in this example we want to use FPP to compute for which values of x the IF-statement computes the absolute value of x. We do this in two steps: (1) compute the weakest precondition; this gives us the set of values of x for which the IF-statement can be successfully executed; (2) check whether the IF- statement computes the absolute value of x for the values determined in step (1).

Step 1: *Input to FPP:*

```
IF x < 0 THEN x := -x; END IF;
--!Post: -128 <= x and x <= 127;
```

Answer of FPP:

```
--> wp: 0 >= 1 + x AND -128 <= -x
-->                AND -x <= 127
-->     OR 0 <= x AND -128 <= x
-->                AND x <= 127
IF x < 0 THEN
   x := -x;
end if;
--!post: -128 <= x AND x <= 127
```

The somewhat clumsy expression for wp can be simplified to: $-127 \le x \le 127$.

Step 2: *Input to FPP:*

```
--!Pre: -127 <= x and x <= 127
--!Pre:           and x = Kx;
IF x < 0 THEN x := -x; END IF;
--!Post: -128 <= x and x <= 127
--!Post: and ((x=Kx and Kx >= 0)
--!Post: or (x = -Kx and Kx<0));
```

Answer of FPP (simplified):

```
--!pre: -127 <= x AND x <= 127
--!pre:           AND x = kx
--> wp: -127 <= x AND x <= 127
--> vc: -127 <= x AND x <= 127
-->      AND x = kx
--> ==>  -127 <= x AND x <= 127
--> Result: proved
IF x < 0 THEN
   x := -x;
END IF;
--!post: -128 <= x AND x <= 127
AND
--!post: (x = kx AND kx >= 0  OR
--!post:  x = -kx AND 0 > kx )
```

We see that the IF-statement computes the absolute value of x.

Exmp 4: *Input to FPP:*

```
x := x+2;
--!Post: x > 50 and x <= 100;
```

Answer of FPP (simplified):

```
--> wp: x >= 49 AND x <= 98
x := x + 2;
--!post: x >= 51 AND x <= 100
```

5 Conclusions

If software engineers want to design programs as other engineers are designing their artifacts a framework for the quantitative analysis of programs is necessary. There exist several

calculi to define the effect of a program in a quantitative manner: e.g. the wp-calculus or the relational calculus of Hehner [Heh 93]. As for other calculations in engineering it is very useful to let the computer do these calculations. In other branches of engineering the calculations necessary are typically numerical ones. For these a lot of tools are available. For the calculations necessary to compute the effect of programs no tools are readily available.

In this paper we have described the Frege Program Prover, an experimental tool which supports the SW engineer in program analysis.

Acknowledgments

The Frege Program Prover has been built by Stefan Kauer and Stefan Knappe [Kna 95, 96].

References

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

Dor 93 Dorf, Richard C. (ed): The Electrical Engineering Handbook. CRC Press, Boca Raton etc. 1993. 0-8493-0185-8

DS 90 Dijkstra, Edsger W.; Scholten, Carel S.: Predicate Calculus and Program Semantics. Springer, New York etc., 1990. 0-387-96957-8

Heh 93 Hehner, Eric C. R.: A Practical Theory of Programming. Springer, New York etc., 1993. 0-387-94106-1

Kna 95 Knappe, Stefan: Berechnung der schwächsten Vorbedingung für eine Teilmenge von Ada. Term Project. Friedrich Schiller Univ., Inst. of Comp. Sci, 1995.Jun.18

Kna 96 Knappe, Stefan: Berechnung von Verifikationsbedingungen für eine Teilmenge von Ada. Diploma Thesis, Friedrich Schiller Univ., Inst. of Comp. Sci, 1996.May.02

Win 96 Winkler, Jürgen F. H.: Some Properties of the Smallest Post-Set and the Largest Pre-Set of Abstract Programs . Friedrich Schiller University, Dept. of Math. & Comp. Sci., Report Math / Inf / 96 / 32 1996.Oct.23

Prof. Dr. Jürgen F. H. Winkler
Friedrich-Schiller University
Institute of Computer Science
D-07740 Jena, Germany
http://www1.informatik.uni-jena.de

Occurred in:

42. Intern. Wissenschaftliches Kolloquium,
Techn. Universität Ilmenau,
22. – 25. 09. 1977
Band 1, p.116 .. 121
ISSN 0943-7207