FRIEDRICH-SCHILLER-
UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik

Jenaer Schriften zur

Mathematik und Informatik

Eingang: 1999.11.09 **Nr. Math / Inf / 99 / 28** **Als Manuskript gedruckt**

A COMPARISON OF THE PROGRAM PROVERS NPPV AND FPP

Stefan Kauer, Jürgen F H Winkler

Friedrich-Schiller University, Institute of Computer Science
D-07740 Jena, Germany
http://www1.informatik.uni-jena.de

A COMPARISON OF THE PROGRAM PROVERS NPPV AND FPP

Stefan Kauer, Jürgen F H Winkler

Friedrich-Schiller University, Institute of Computer Science, D-07740 Jena, Germany
http://www1.informatik.uni-jena.de

This paper compares two experimental systems for the mechanical verification of small programs: NPPV (New Paltz Program Prover) and FPP (Frege Program Prover). Both systems are based on the wp-semantics of the constructs they support. NPPV can also deal with partial correctness of loops, whereas FPP always tries to prove loop termination. NPPV supports a subset of Pascal and FPP supports a subset of Ada. Both tools use automatic theorem proving. NPPV uses a simple rewrite system, whereas FPP uses an extended version of the theorem prover Analytica.
The comparison is done by trying to prove 26 examples from various sources, but mainly from the NPPV distribution. 23 of the 26 examples are correct and can in principle be proved. The main result is that NPPV is able to prove only one of the 22 examples which are appropriate for it. The essential limitation seems to be the theorem prover used in NPPV. Since FPP only supports total correctness only 17 of the 24 correct examples are appropriate for FPP. FPP proves 14 of those 17.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—assertion checkers, *correctness proofs*; F.3.1 [**Logics and Meanings of Programs**] Specifying and Verifying and Reasoning about Programs—*assertions, mechanical verification, pre- and post-conditions*

General Terms: Verification

Additional Key Words and Phrases: program correctness, verification conditions, automatic program prover

1 Introduction

This paper compares two experimental systems for the mechanical verification of small programs: NPPV (New Paltz Program Prover) and FPP (Frege Program Prover). Both systems are based on the wp-semantics of the constructs they support. NPPV can also deal with partial correctness of loops, whereas FPP always tries to prove loop termination. NPPV supports a subset of Pascal and FPP supports a subset of Ada. Both tools use automatic theorem proving. NPPV uses a simple rewrite system, whereas FPP uses an extended version of the theorem prover Analytica.

The comparison is done by trying to prove 26 examples from various sources, but mainly from the NPPV distribution. 23 of the 26 examples are correct and can in principle be proved. The main result is that NPPV is able to prove only one of the 22 examples which are appropriate for it. The essential limitation seems to be the theorem prover used in NPPV. Since FPP only supports total correctness only 17 of the 23 correct examples are appropriate for FPP. FPP proves 14 of those 17.

The paper is organized as follows. Section 2 gives a very short introduction into program verification, especially verification based on weakest preconditions. In section 3 we describe the main characteristics of NPPV and in section 4 those of FPP. The results of the comparison based on 26 examples are presented in section 5. The appendix contains the verification protocols of all 26 examples for both NPPV and FPP.

2 Program Verification

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 the case that a software engineer can calculate the effect of a simple program or program fragment in a quantitative way. Examples of such computations are:

- computing the weakest precondition for a given program fragment PF and a given postcondition Post
- computing the strongest postcondition for a given program fragment PF and a given postcondition Post
- checking whether a triple {Pre} PF {Post} is consistent
- computing counterexamples if a triple {Pre} PF {Post} is not consistent

The Frege Program Prover (FPP) and the New Paltz Program Verifier (NPPV) are experimental systems to support the SW engineer in such calculations.

Program verification usually deals with the question whether a triple

$$\{Pre\} \quad PF \quad \{Post\} \tag{1}$$

is consistent. This can be formally defined as:

$$[\ Pre \ \Rightarrow \ wp(PF, Post)\ ] \tag{2}$$

if we use wp(weakest precondition) for the characterization of program semantics. The square brackets (”[ ]“) are an abbreviation for the universal quantification over the program variables (”in all states“) [DS 90]. There exist similar mechanisms for the characterization of the semantics of programs e.g. the combination of wp and wlp (weakest liberal precondition) [DS 90] or pre- and post-sets [Win 96]. Since the two tools, which are compared in this paper, both use wp for the characterization of semantics we use in the discussion on verification also wp. It is not our intention to explain the theory of program verification in this report; more details on program verification are given in [DS 90; Gri 83; KW 97; KW 99a; Win 97]. In general (2) is a theorem, and to show the consistency of (1) is the same as to try to prove (2). Such theorems are often also called verification conditions (VC).

Examples:

$\{v>0\}$ v:=v+1; $\{v>4\}$

$\equiv$ $[\ v>0 \ \Rightarrow\ wp(”v:=v+1;“, v>4)\ ]$

$\equiv$ $[\ v>0 \ \Rightarrow\ v+1 \in type(v) \ \wedge\ v \geq 4\ ]$

$\equiv$ False.

$\{v>0 \ \wedge v+1 \in type(v)\ \}$ v:=v+1; $\{v>1\}$

$\equiv$ $[\ v>0 \ \wedge v+1 \in type(v) \ \Rightarrow\ wp(“v:=v+1;“, v>1)\ ]$

$\equiv$ $[\ v>0 \ \wedge v+1 \in type(v) \ \Rightarrow\ v+1 \in type(v) \ \wedge\ v+1 > 1\ ]$

$\equiv$ True.

$\{\ -127 \ \leq\ x\ \leq\ 127 \ \ \wedge\ \ x = x\_i\ \}$
IF x < 0 THEN x := -x; END IF;
$\{\ 0 \leq\ x\ \leq 127 \ \wedge\ (\ x = x\_i \vee\ x = -x\_i\ \ )\ \}$

$\equiv$ $[\ -127 \ \leq\ x\ \leq\ 127 \ \ \wedge\ \ x = x\_i \ \Rightarrow$
wp(”IF x < 0 THEN x := -x; END IF;“, $0 \leq\ x\ \leq 127 \ \wedge (\ x = x\_i \vee\ x = -x\_i\ )\ )\ ]$

$\equiv$ $[\ -127 \ \leq\ x\ \leq\ 127 \ \ \wedge\ \ x = x\_i \ \Rightarrow$
$x < 0 \ \ \wedge\ \ 0 \leq\ -x\ \ \leq 127\ \wedge\ \ (\ -x = x\_i \vee\ -x = -x\_i\ )\ \vee$
$0 \leq\ x\ \ \wedge\ \ 0 \leq\ x\ \ \leq 127\ \wedge (\ x = x\_i \vee\ x = -x\_i\ )\ \ ]$

$\equiv$ $[\ -127 \ \leq\ x\ \leq\ 127 \ \ \wedge\ \ x = x\_i \ \Rightarrow$
$x < 0 \ \ \wedge\ -127 \leq\ x\ \ \leq 0\ \ \wedge\ \ (\ x = x\_i \ \vee\ x = -x\_i\ \ )\ \vee$
$0 \leq\ x\ \ \leq 127\ \wedge (\ x = x\_i \ \vee\ \ x = -x\_i\ )\ \ ]$

$\equiv$ $[\,-127 \leq x \leq 127 \;\wedge\; x = x\_i \;\Rightarrow$
$(-127 \leq x < 0 \;\vee\; 0 \leq x \leq 127\,) \;\wedge\; (\,x = x\_i \;\vee\; x = -x\_i\,)\;]$

$\equiv$ $[\,-127 \leq x \leq 127 \;\wedge\; x = x\_i \;\Rightarrow$
$(-127 \leq x \leq 127 \;\wedge\; x = x\_i\,) \;\vee\; (\,-127 \leq x \leq 127 \;\wedge\; x = -x\_i\,)\;]$

$\equiv$ True.

In general the assertions (i.e. precondition and postcondition) contain two kinds of variables: (1) program variables which occur also in the program, and (2) specification variables, which occur in the assertions only. Such variables are called ”ghost variables“ in [Bac 86: 86]. The specification variables are typically used to refer in the postcondition to the initial value of a program variable, or vice versa. In this paper we use the convention that the identifiers of such specification variables are obtained by appending the suffix ”_i“ for the initial value or the suffix ”_f“ for the final value to the identifier of the corresponding program variable. x_i denotes the initial value of the program variable x and abc_f the final value of the program variable abc. It may even be better to use uppercase letters or such identifiers for specification variables which are syntactically not allowed for program variables, e.g. in Ada identifiers containing ‘\’. Examples of such specification variables are ”X_i\“, ”X\i“, which could denote the initial value of X and ”X_f\“, ”X\f“ for the final value of X.

The use of these specification variables leads to somewhat clumsier assertions. In informal treatment they are therefore often not used [Bac 86: 150 f.; Gri 83: 100]. Quite often authors use very weak specifications in the verification of much stronger programs. A typical case is that the specification does not state that certain variables are not to be changed. This leads generally to positive verification results. Nevertheless, we think that such weak specifications are not appropriate in general. In program development the specification is typically developed first and then used as an input to program development. If the specification allows also for a very simple program, why should we then develop a more complicated program at all ?

As we see in these examples, verification leads to a lot of formula manipulation. It is rather tedious and error prone to do this by hand. As with numeric computations the computer can also perform symbolic computations very much better than human beings. Such tools can be called ”program verifiers“ or ”program provers“. If we give the third example to the Frege Program Prover (FPP) we obtain the following result:

Input to FPP

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

Output from FPP (simplified):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.08.30, 16:04
The answer to your query is:


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

The automatic verification of such an annotated program consists of two main steps:

- generation of the VCs. This is a straightforward task.
- the automatic proof of the VCs generated in the first step.

If such proofs are done by a tool T this tool contains as one of its components an automatic theorem prover. The capabilities of this theorem prover then limit the capabilities of T. The ATP tries to prove the VCs which are formally theorems in a first order predicate logic. Theorem proving is not as straightforward as the generation of the VCs. Therefore, the ATP is the hard part of program proving.

The last example in this section shows the verification of a FOR-loop using FPP [KW 97].

Input to FPP:

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

Output from FPP (simplified):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.08.30, 16:24
The answer to your query is:

--!pre       : r = 1 AND 0 <= n AND n <= 20 AND n = n_i
--!post      : r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i
--!inv       : r = Factorial(i) AND 0 <= n AND n <= 20 AND n = n_i
-->functionality ------------------------------------------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :     1 <= n AND r = 1 AND 0 <= n AND n <= 20 AND n = n_i
-->            ==> r = 1 AND 0 <= n AND n <= 20 AND n = n_i
-->Result    : proved
-->induction :     r = Factorial(i-1) AND 1 <= n AND n <= 20 AND n = n_i
-->            ==> i*r = Factorial(i) AND 0 <= n AND n <= 20 AND n = n_i
-->Result    : proved
-->final     :     r = Factorial(n) AND 1 <= n AND n <= 20 AND n = n_i
-->            ==> r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i
-->Result    : proved
-->null loop :     n = 0 AND r = 1 AND n = n_i
-->            ==> r = Factorial(n) AND -2**63 <= r AND r <= 2**63-1 AND n = n_i
-->Result    : proved
FOR i IN 1 .. n  LOOP
   r := r * i;
END LOOP;
```

3 NPPV: The New Paltz Program Verifier

NPPV [GS 98, Gum 99b] is an automatic program verifier for a subset of Pascal. The input consists of a program with assertions. NPPV generates the VCs and tries to prove them. The VCs are generated using the wp calculus (for assignment, if and sequence) [Gri 83]. The VCs for WHILE-loops and for FOR-loops are generated as described in [Gri 83; Hoa 72]. No range checks are generated. The output consists of the VCs and the result of the prover. If the proof succeeds, the output says ”succeeded“, otherwise it says ”Remains to prove“ and lists the VCs which could not be proved. A program is consistent with its specification, if the proofs of all verification conditions succeed. Unfortunately, in the examples in the Appendix (see also Table 5.2) most of the interesting VCs remain to be proved. The output can be saved in a log file called ”session.log”, i.e. the name of the output file is independent of the name of the input file.

The programming language supported contains the types integer with range -32768..32767 and array and the statements assignment, IF, FOR-loop and WHILE-loop. The assertions are Boolean expressions of

Pascal, which implies that quantifiers and implication are not allowed. Loops require an invariant. Assertions and invariants are comments. WHILE-loops may be annotated with a termination function. The termination function occurs immediately after the keyword ”DO“ and is enclosed in square brackets. Without such a function only partial correctness can be proved.
Identifiers starting with small letters are variables. Identifiers starting with capital letters are constants. That means that for example n:=5 is allowed, whereas N := 5 (i.e. assignment in general) is not allowed. This is checked by NPPV. There is no explicit distinction between program and specification variables, but these constants have the essential properties of specification variables and are used as such in the examples in the appendix.

Example:

Input to NPPV:

```
{v > 0}
BEGIN
   v := v+1
END
{v > 4}
```

Output from NPPV:

```
Generating verification conditions...
O.K.

=====================================
=== Verification Condition No.: 1 ===

v>0
         ==>
                v+1>4

--------- Remains to prove  ---------
0<v
         ==>
                4<v+1
=====================================
```

NPPV can be obtained from http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html. It runs on DOS or in a DOS window of WindowsNT or OS/2.

4 FPP: The Frege Program Prover

The Frege Program Prover (FPP; http://www1.informatik.uni-jena.de/FPP/FPP-main.htm) is a tool to perform calculations in the area of program analysis (see sect. 1). It is based on the wp-calculus and a small subset of Ada (types integer and Boolean; assignment, IF, CASE, FOR and WHILE). Assertions are written as special comments (--! . . .) and are essentially Boolean expressions; additionally to Ada quantifiers and the implication are allowed. The syntax of the input language is given in http://www1.informatik.uni-jena.de/FPP/fpp-synt.htm. For FOR-loops an invariant has to be given and for WHILE-loops an invariant and a termination function. Therefore, FPP deals only with total correctness of loops and not with partial correctness.

Since we are not working in automatic theorem proving FPP uses a theorem prover from another party: Analytica [CZ 92] with some modifications and extensions implemented by S. Kauer. Due to the difficulty of automatic theorem proving the user of FPP may encounter a situation in which FPP cannot prove a certain VC, runs out of memory during proving or runs for a very long time. Such behavior is not completely different from that of a human theorem prover who may also not be able to prove a certain theorem. The capabilities of FPP just reflect the capabilities of the built-in ATP.

One example for such a behavior is example 8 (fastmultty). The prover transforms the VC into DNF (disjunctive normal form) and splits this transformed VC into an equivalent conjunction of clauses. A clause is an implication of a conjunction of atoms and a single atom, similar to a clause in PROLOG. But unlike in PROLOG, the clauses here may contain quantifiers. In the worst case, the transformation into DNF is of exponential complexity, in space as well as in time [Sch95: 30].

A further limitation lies in the state of the wp-calculus. For loops e.g. a correctness proof is usually done using an invariant and (for the WHILE loop) a termination function. Invariant and termination function usually are provided by the user. If the invariant is too sharp it may not be possible to prove the consistency of [Pre $\Rightarrow$ wp(PF, Post)] even if it is consistent. Therefore, tools for proving the consistency of specified programs can be improved by developing new rules for the computation of wp. This has been done in [Kau 99], which contains two new methods for the computation of wp for loops. The first method computes for certain FOR-loops automatically an invariant, and the second computes automatically the weakest precondition of certain WHILE-loops.

FPP is implemented as a WWW application, i.e. it can be used interactively over the net.

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 4.1: wp(”v:=v+1;“, v>4)
   $\equiv$ v+1 $\in$ type(v) $\wedge$ v+1 > 4
   $\equiv$ v+1 $\in$ type(v) $\wedge$ v $\geq$ 4.

   type(v) is the value set of v, i.e. the set of admissible values as defined by the type of v.

b) *Check the correctness of a program* (fragment) wrt a specification (Pre, Post): 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. As mentioned in sect. 2, this consistency is defined as [Pre $\Rightarrow$ wp(PF, Post)]

   Example 4.2: {v>0} v:=v+1; {v>4}
   $\equiv$ [v>0 $\Rightarrow$ wp(”v:=v+1;“, v>4)]
   $\equiv$ [v>0 $\Rightarrow$ v+1 $\in$ type(v) $\wedge$ v $\geq$ 4]
   $\equiv$ False.

If we apply the FPP to these examples we obtain:

Example 4.1 *Input to FPP:*

```
v := v+1;
   --!Post: v > 4 and -100 <= v  and v <= 100;
```

Since FPP currently knows nothing about value sets of variables the range constraint on v is expressed explicitly in the postcondition (`-100 <= v  and v <= 100`).

Output from FPP:

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1998.03.06, 14:48

The answer to your query is:

--> wp    : (1+v >= 5 AND -100 <= 1+v AND 1+v <= 100)
v := v + 1;
--!post   : (v >= 5 AND -100 <= v AND v <= 100)
```

The lines containing results computed by FPP begin with the character combination ”-->“. In the output we see that FPP has computed the weakest precondition and that the postcondition had been given by the user.

Example 4.2 *Input to FPP:*

```
      --!Pre: v > 0;
   v := v+1;
      --!Post: v > 4 and -100 <= v  and v <= 100;
```

Output from FPP (simplified):

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1998.03.06, 15:00
The answer to your query is:

--!pre : (v >= 1)
--> wp : (1 + v >= 5 AND -100 <= 1 + v AND 1 + v <= 100)
--> vc : (v >= 1 ==> 1+v >= 5 AND -100 <= 1+v AND 1+v <= 100)
--> Result: not proved
--> Incorrectness condition: (3 >= v AND v >= 1)
v := v + 1;
--!post: (v >= 5 AND -100 <= v AND v <= 100)
```

In this example the program fragment and the given specification are not consistent. Therefore, FPP gives the answer ”not proved“. It gives further an ”incorrectness condition“ which characterizes those states which fulfill the precondition but from which the program is not guaranteed to finish in a state of the postcondition.

In such a case it is additionally useful if the user is presented a concrete counterexample. A method to compute counterexamples in certain cases has been developed in [Kau 99]. Up to now, this method has not been incorporated into FPP. In this very simple example we see that   v=1   is a counterexample; but in more complicated cases it is not so easy to derive a counterexample from the incorrectness condition.

The following example shows how loops are annotated i.e. especially how the invariant and the termination function are specified by the user. The program computes the gcd of two numbers using Euclid’s algorithm..

Example 4.3 ***Input to FPP:***

```
   --!pre: i>0 and j>0 and i=i_i and j=j_i;
   --!post: i=j and i=GGT(i_i,j_i);
   --!inv: i>0 and j>0 and GGT(i,j) = GGT(i_i,j_i);
   --!term: i+j;
WHILE i /= j LOOP
   IF i>j
   THEN i := i-j;
   ELSE j := j-i;
   END IF;
END LOOP;
```

For loops all annotations are given before the beginning of the loop. In this example precondition, postcondition, invariant and termination function are given. FPP then tries to show the validity of

$$[\text{ pre} \Rightarrow \text{inv }] \wedge [\text{ cond} \wedge \text{inv} \Rightarrow \text{wp(body, inv) }] \wedge [\neg\text{cond} \wedge \text{inv} \Rightarrow \text{post }] \wedge$$
$$[\text{ cond} \wedge \text{inv} \Rightarrow \text{term} > 0] \wedge [\text{ cond} \wedge \text{inv} \Rightarrow \text{wp(”T:=term; body“, term} < \text{T) }]$$

which is the classical consistency condition for WHILE-loops.

***Output from FPP** (slightly edited):*

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.01, 16:39
The answer to your query is:
```

```
--!pre       : (i >= 1 AND j >= 1 AND i = i_i AND j = j_i)
--!post      : (i = j AND i = GGT(i_i,j_i))
--!inv       : (i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
--!term      : (j + i)
-->functionality --------------------------
-->initial   :     (i >= 1 AND j >= 1 AND i = i_i AND j = j_i)
-->            ==> (i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->Result    : proved
-->induction :     (i /= j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==>          (i > j) AND (j >= 1) AND (GGT(i,j) = GGT(i_i,j_i))
-->                 OR      (i < j) AND (i >= 1) AND (GGT(i,j) = GGT(i_i,j_i))
-->Result    : proved
-->final     :     (i = j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==> (i = j AND i = GGT(i_i,j_i))
-->Result    : proved
-->termination --------------------------
-->initial   :     (i /= j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==> (j + i >= 1)
-->Result    : proved
-->induction :     (i /= j AND i >= 1 AND j >= 1 AND GGT(i,j) = GGT(i_i,j_i))
-->            ==> (i >= 1 + j AND j + i >= 1 + i OR i < 1 + j AND j + i >= 1 + j)
-->Result    : proved
WHILE i /= j LOOP
   IF i > j THEN
      i := i - j;
   ELSE
      j := j - i;
   END IF;
END LOOP;
```

All five conjuncts of the consistency condition have been proved and therefore, the LOOP and the specification are consistent.

In two cases auxiliary variables are generated during the proof process. Auxiliary variables are neither program nor specification variables.

The first case occurs with nested loops. The inner loop is treated as a procedure call. Hence it is proved by a proof rule for procedure calls [Kau 99]. This proof rule generates a formula with a universal quantifier. The bound variables of these quantifiers are auxiliary variables. The following example shows a nested loop:

```
   --!pre: a>=0 and b>=0;
p := 0;
   --!pre: p=0 and a>=0 and b>=0;
   --!post: p=b*a;
   --!inv: p=i*a and a>=0;
for i in 1..b loop
      --!pre: p = (i-1)*a and a>=0;
      --!post: p = i*a;
      --!inv: p = (i-1)*a+j;
   for j in 1..a loop
         p := p mod 1;
   end loop;
end loop;
```

This nested loop computes the product of a and b by multiple addition. The addition is computed by multiple application of the successor function.

The output from FPP is:

```
--!pre       : (a >= 0 AND b >= 0)
--> wp       : (a >= 0 AND b >= 0)
--> vc       : (True)
--> Result: proved
p := 0;
--!pre       : (p = 0 AND a >= 0 AND b >= 0)
--!post      : (p = a*b)
--!inv       : (p = a*i AND a >= 0)
```

```
-->functionality ---------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :(1 <= b AND p = 0 AND a >= 0 AND b >= 0 ==> p = 0 AND a >= 0)
--> Result    : proved
-->induction :     (1 <= b AND p = a*(-1 + i) AND a >= 0)
-->            ==>          (1 <= a)
-->                     AND (p = a*(-1 + i))
-->                     AND (a >= 0)
-->                     AND (forall(($0)),
-->                                  (p = a*(-1 + i) AND a >= 0 ==> $0 = a*i)
-->                              ==> ($0 = a*i AND a >= 0))
-->                 OR (1 >= 1 + a AND p = a*i AND a >= 0)
--> Result    : proved
-->final     :(1 <= b AND p = a*b AND a >= 0 ==> p = a*b)
--> Result    : proved
-->null loop :(1 >= 1 + b AND p = 0 AND a >= 0 AND b >= 0 ==> p = a*b)
--> Result    : proved
FOR i IN 1 .. b  LOOP

   --!pre       : (p = a*(-1 + i) AND a >= 0)
   --!post      : (p = a*i)
   --!inv       : (p = j + a*(-1 + i))
   -->functionality ---------------------------
   -->func      : (initial AND induction AND final AND null loop)
   -->initial   :(1 <= a AND p = a*(-1 + i) AND a >= 0 ==> p = a*(-1 + i))
   --> Result    : proved
   -->induction :(1<=a AND p= -1 + j + a*(-1 + i) ==> 1 + p = j + a*(-1 + i))
   --> Result    : proved
   -->final     :(1 <= a AND p = a + a*(-1 + i) ==> p = a*i)
   --> Result    : proved
   -->null loop :(1 >= 1 + a AND p = a*(-1 + i) AND a >= 0 ==> p = a*i)
   --> Result    : proved
   FOR j IN 1 .. a  LOOP
      p := p + 1;
   END LOOP;
END LOOP;
```

The induction part of the outer loop contains

```
          (forall(($0)), (p = a*(-1 + i) AND a >= 0 ==>
                         $0 = a*i) ==> ($0 = a*i AND a >= 0))
```

The variable $0 is an auxiliary variable and starts with a '$' in order to prevent a clash between program or specification variables and auxiliary variables.

The second case, in which auxiliary variables occur, is through skolemization during the generation of falsification conditions (FC). FCs are generated, whenever a VC cannot be proved. Example:

Input to FPP:

```
--!pre: (exists x:0 <= x and x <10: x<y);
y := y-1;
   --!post: y>0;
```

Output from FPP:

```
--!pre       : ((exists x: 0 <= x AND 10 >= 1 + x AND y >= 1 + x))
--> wp       : (-1 + y >= 1)
--> vc       : ((exists x: 0 <= x AND 10 >= 1 + x AND y >= 1 + x) ==> -1 + y >= 1)
--> Result: not proved
--> fc       : (2 - y >= 1 AND 9 - $07 >= 0 AND -1 + y - $07 >= 0 AND $07 >= 0)
y := y - 1;
--!post      : (y >= 1)
```

The example program is not correct and therefore, its correctness cannot be proved. FPP then generates an FC, which contains the auxiliary variable $07. The FC is a system of equalities and inequalities. The solution of an FC is a counterexample, which explicitly shows, that the program is incorrect [Kau 99]. The computation of the solution of the FC is not yet implemented. In this example one (and the only) solution is y = 1 and \$07 = 0, which yields the counterexample y = 1 and x = 0 as an initial state. A larger example, in which both kinds of auxiliary variables occur, is example 7 (fastmult) in the appendix.

5 Comparison of NPPV and FPP

NPPV and FPP are similar systems. Both are based on a subset of an imperative programming language extended by assertions (pre- and post-conditions, invariants and termination functions). Both consist of a verification condition generator (VCG) and an automatic theorem prover. The input of both consists of a program with assertions, and the output of both contains the VCs generated by the VCG and the result of the ATP.

The differences between NPPV and FPP lie in the details, which are listed in table 5.1.

	NPPV	FPP
programming language	subset of Pascal	subset of Ada
assertion language	subset of Pascal expressions, enclosed in { }; true is expressed by {}	subset of Ada expressions extended with *quantifiers*, *implication* and the additional functions abs, min, max, ggt, sum, factorial, fib
form of assertions	{}comments	special comments: --!
multiline assertions	supported	supported
supported types	integer, array with integer index type and integer component type	integer and Boolean
supported statements	assignment, IF, FOR- and WHILE loop	NULL, assignment, IF, CASE, FOR and WHILE loop
proof of loops	only invariant required, termination function for WHILE loops optional, so that also partial correctness can be proved	precondition, postcondition, invariant and for WHILE loops termination function necessary, so that only total correctness can be proved
output	optional in a file: session.log; output contains only verification conditions and results	in a file that has the same name as the input file, but a different extension; output contains the statements, the VCs and the result together
usage	local	local or via WWW
pretty printing	not supported	supported
simplification of expressions	not performed	performed to a certain extent
computing wp	not possible	possible
theorem proving	possible e.g. with x := x	possible e.g. with null statement
implementation language	Visual Prolog	Ada, C and Mathematica
proving power	only trivial	higher than NPPV
automatic theorem prover	simple rewrite system	mexana, an extension of Analytica

Table 5.1. Properties of NPPV and FPP

Table 5.2 contains the results, that were obtained when trying to verify a series of examples with both NPPV and FPP. The 23 examples come from two sources. Those with the indication ”Gumm“ in the column ”Source“ are delivered with the NPPV system and those with the indication ”Kauer“ have been created by S. Kauer. The complete results for the 23 examples are contained in the Appendix of this report.

The right part of Table 5.2 contains the essential results when both provers try to prove the examples. There are examples which are not appropriate for one of the program provers; e.g. example 2 (”array“) is not appropriate for FPP, because FPP does not (yet) support the data type ”array“. Example 16 (”linsearch“) is not appropriate for NPPV, because NPPV does not support the use of quantifiers. For such inappropriate examples no results can be given. For the appropriate examples we give three results:

- whether it could be proved (all examples are consistent)
- the number of VCs which have been generated
- the number of VCs which have been proved

No.	Program	Remark	Source	FPP			NPPV		
				proved	#VC	#proved	proved	#VC	#proved
1	abs	no abs function in NPPV	Kauer	YES	1	1	NO	2	1
2	array	assignment to arrays not supported in FPP	Gumm	n.a.			NO	1	0
3	assrek	no termination function	Gumm	n.a.			NO	4	1
4	factfor		Gumm	YES	5	5	NO	5	2
5	factforty		Gumm / Kauer	YES	6	6	NO	5	1
6	fastmul	no termination function	Gumm	n.a.			NO	6	4
7	fastmult		Gumm	NO	10	7	NO	14	7
8	fastmultty	too many clauses[1]	Gumm / Kauer	NO	11	7	NO	14	7
9	fibo	no termination function	Gumm	n.a.			NO	4	2
10	fibot		Gumm	YES	6	6	NO	6	4
11	fibotty		Gumm / Kauer	NO	6	5	NO	6	3
12	gauss	no termination function	Gumm	n.a.			NO	4	3
13	gausst		Gumm	YES	6	6	NO	6	5
14	gaussty		Gumm / Kauer	YES	6	6	NO	6	4
15	linrek	no termination function	Gumm	n.a.			NO	7	2
16	linsearch	no quantifiers allowed in NPPV’s assertions	Kauer	YES	5	5	n.a.		
17	nested_for		Kauer	YES	9	9	NO	8	5
19	quad		Kauer	YES	5	5	NO	5	3
20	root		Kauer	YES	5	5	NO	5	4

[1] In this case no output was generated after 24 hours computing time. A closer look into the execution of the theorem prover revealed, that too many clauses were generated (see sect. 4).

21	swap1		Gumm	YES	1	1	NO	1	1
22	swap2		Gumm	YES	1	1	YES	1	1
24	swap2ty2		Gumm / Kauer	YES	3	3	NO	3	1
26	cube		[GH 99]	YES	4	4	NO	5	3
				14 (17)	90	82	1 (22)	118	64

Table 5.2. Results for 23 Programs

The last line of Table 5.2 contains a quantitative summary of the 23 examples. It is somewhat surprising that NPPV is not able to prove those examples which are delivered with the distribution of the system. FPP proves 14 of the 17 examples which are appropriate and NPPV proves 1 example of the 22 appropriate ones. This is the example 22 (swap2) which consists of three rather simple assignment statements. NPPV is able to generate the necessary VCs, even rather complicated ones as e.g. in example 2 (array). But in almost all examples it is not able to perform the necessary proofs. The reason for this is that NPPV uses only a simple rewrite system as theorem prover [Gum 99c]. Examples of very simple theorems which NPPV cannot prove are:

Example 12, VC4:	$x < 0 \Rightarrow x+1 \le 0$
Example 17, VC6:	$0 \le a \Rightarrow (i-1)*a+a = a*i$
Example 19, VC5:	$0 \le i \wedge i+1 \le k \Rightarrow 2*i+1 = i+i+1$

For WHILE-loops FPP only uses the proof rule for total correctness which uses both an invariant and a termination function. Due to this reason those examples, which contain a WHILE-loop and no termination function (examples 3, 6, 9, 12) are not appropriate for FPP. On the other hand, those examples are appropriate for NPPV. But NPPV cannot prove them.

In one case FPP is not able to complete the proof, because too many clauses are generated during the proof (example 8).

The appendix contains three more examples which are not contained in Table 5.2. Example 23 (swap2ty) is not correct, i.e. the specification and the program are not consistent. Therefore it is correct that none of the tools can prove swap2ty. In the examples 18 (proof) and 25 (swap3) the program provers are used to do formula manipulation only; the goal of the example is to generate the verification condition. The specification does not contain enough information to prove the example. Example 18 is not appropriate for FPP.

6 Conclusion and Outlook

We have presented a comparison of two automatic program provers, NPPV and FPP. Both provers use a similar approach to the program proving and support a similar language. The main difference is the power of the theorem prover. That of NPPV is rather weak because it cannot prove almost all of the consistent examples. FPP uses a somewhat stronger theorem prover and can therefore prove 14 of those 17 examples which are appropriate for FPP.

Improvements to FPP are possible in different directions:

a) Use of a larger language. Currently FPP uses a quite small subset of Ada. Work is underway on FPP-2 which will support a larger subset of Ada. New elements are declarations, arrays, records and procedures. FPP-2 will especially check the type conditions automatically and therefore simplify the task of the user.

b) Use of more methods for the generation of verification conditions. FPP-2 will support a new proof rule for the FOR-loop [Win 98], a method for the automatic determination of an invariant for FOR-loops

[Kau 99; KW 99b], a method for the computation of a concrete counterexample for a verification condition which could not be proved [Kau 99], and a method for the direct verification of certain WHILE-loops, which does not require an invariant [Kau 99].

c) Use of a stronger theorem prover. Since we are not working in the area of theorem proving, we can only use those provers which are available. In the beginning FPP-2 will use the same theorem prover as FPP-1.

d) Adding an interactive mode in which the user can give additional hints to the theorem prover.

7 References

Bac 86	Backhouse, Roland C.: Program Construction and Verification. Prentice Hall, New York etc., 1986. 0-13-729146-9
CZ 92	Clarke, E.; Zhao, X.: Analytica - A Theorem Prover in Mathematica. International Conference on Artificial Intelligence and Symbolic Mathematical Computation. AISM C-3, Steyr 1996, 21 - 37
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
GH 99	Hehner, Eric C. R.; Gravell, Andrew M.: Refinement Semantics and Loop Rules. In: FM’99, Vol. II, 1497..1510. LNCS 1709, Springer, Berlin etc., 1999. 3-540-66588-9
Gri 83	Gries, David: The Science of Programming. Springer, New York. etc., 1983. 0-387-90641-X
GS 98	Gumm, Heinz-Peter; Sommer, Manfred: Einführung in die Informatik. R. Oldenbourg Verlag. München 1998. 3-486-24422-1
Gum 99a	Gumm, Heinz-Peter: Generating algebraic laws from Imperative Programs. Theoretical Computer Science 217(2): 385-405 (1999)
Gum 99b	http://www.mathematik.uni-marburg.de/~gumm/NPPV/nppv.html, 10 Mar. 1999
Gum 99c	Gumm, Heinz-Peter: Letter to J F H Winkler. 1999.Oct.26
Hoa 72	Hoare, C.A.R.: A Note on the FOR Statement. BIT 12,3 (1972) 334..341
Kau 99	Kauer, Stefan: Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. Dissertation, Friedrich-Schiller-Universität Jena, 1999.
KW 97	Winkler, Jürgen F. H.; Kauer, Stefan: Proving Assertions is also Useful. SIGPLAN Notices 32,3 (1997) 38..41
KW 99a	Kauer, Stefan; Winkler, Jürgen F. H.: FPP: An Automatic Program Prover for Ada Statements. GI FG 2.1.5. Ada: Workshop ”Objektorientierung und sichere Software mit Ada“. Karlsruhe 1999.Apr.21-22
KW 99b	Kauer, Stefan; Winkler, Jürgen F. H.: Automatic Genaration of Invariants for FOR-Loops Based on a New Proof Rule. (in preparation)
NW 89	Winkler, J.F.H.; Nievergelt, J.: Wie soll die Fakultätsfunktion programmiert werden? Informatik-Spektrum 12,4 (1989) 220..221.
Sch95	Schöning, Uwe: Logik für Informatiker. Spektrum Akademischer Verlag. Heidelberg 1995. 3-86025-684-x
Win 90	Winkler, J F H: Functions not equivalent. Letter to the editor, IEEE Software 7, 3 (1990) 10
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
Win 97	Winkler, J. F. H.: The Frege Program Prover FPP. 42. Internationales Wissenschaftliches Kolloquium, TU Ilmenau, 1997, 116 .. 121
Win 98	New Proof Rules for FOR-loops. Friedrich Schiller University, Dept. of Math. & Comp. Sci., Report Math / Inf / 98 / 13 1998.Nov.07

Appendix

The appendix contains the listings of the 26 examples. For each example we give a short explanation what the effect of the program should be and the listings of the verification with NPPV and with FPP.

As can be seen in Table 5.2 most of the examples are taken from the NPPV distribution. Since NPPV runs under DOS the file names which are the same as the names of the examples are limited to 8 characters. This leads sometimes to rather terse names.

The examples are ordered alphabetically and numbered. Every example contains a very short informal description, the program in NPPV syntax with Output of NPPV and the equivalent program in FPP syntax with Output of FPP. We also give an explanation for the conversion from NPPV to FPP, if the conversion is possible. If the conversion is not possible, we also give a reason. When type checking is possible, we sometimes repeat the example with type checking assertions. We omit comments to save space. The input files for NPPV require the extension “ver”. Since NPPV runs under DOS, the file names must be at most 8 characters long and therefore they may be somewhat cryptic.

The NPPV examples in table 5.2, which are marked with “Gumm“, are from the original NPPV distribution. In these examples the specifications are sometimes rather weak (e.g. 1, 4, 5, 12).

1. abs Compute y as the absolute value of the given value x.

Input to NPPV

There is no abs function in NPPV, therefore the abs condition is expressed explicitely.

```
{-127 <=x and x <= 127}
BEGIN
if x<0 then y := -x else y := x
END
{ y <= 127 and (x <= 0 and y = -x or x >= 0 and y = x) }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

-127<=x AND x<=127 AND x<0
         ==>
                -x<=127 AND (x<=0 AND -x=-x OR x>=0 A
ND -x=x)

--------- Remains to prove  ---------
-127<=x AND x<=127 AND x<0
         ==>
                -x<=127 AND x<=0 OR x=-x AND -x<=127
AND 0<=x
=====================================
=====================================
=== Verification Condition No.: 2 ===

-127<=x AND x<=127 AND NOT x<0
         ==>
                x<=127 AND (x<=0 AND x=-x OR x>=0 AND
 x=x)

=========  Proof succeeded  =========
=====================================
```

NPPV generates two VCs but can prove only one of them.

Input to FPP

```
   --!pre: -127 <= x and x <= 127 and x = x_i;
IF x<0
THEN y := -x;
ELSE y := x;
END IF;
   --!post: y <= 127 and y = Abs(x) and x = x_i;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.02, 8:21
The answer to your query is:

--!pre       : (-127 <= x AND x <= 127 AND x = x_i)
--> wp       :     (0 >= 1 + x AND -x <= 127 AND -x = Abs(x) AND x = x_i)
-->            OR (0 <= x AND x <= 127 AND x = Abs(x) AND x = x_i)
--> vc       :      (-127 <= x AND x <= 127 AND x = x_i)
-->            ==>     (0 >= 1 + x AND -x <= 127 AND -x = Abs(x) AND x = x_i)
-->                 OR (0 <= x AND x <= 127 AND x = Abs(x) AND x = x_i)
--> Result: proved
IF x < 0 THEN
   y := -x;
ELSE
   y := x;
END IF;
--!post      : (y <= 127 AND y = Abs(x) AND x = x_i)
```

2. array

Somewhat complicated assignment to an array component.

Input to NPPV

```
{ a[1] = 1 and a[5] = X }
  a[a[1]] := 5
{ a[a[1]] = 5 }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

a[1]=1 AND a[5]=X
     ==>
           a{a[1] --> 5}[a{a[1] --> 5}[1]]=5
--------- Remains to prove ---------
X=5
=====================================
```

FPP: assignment to array elements is not yet supported. Therefore, this example cannot be done by FPP.

3. assrek

assrek computes iteratively the recursively defined function

f(x) := if P(x) then g(x) else f(r(x)) [Gum 99b]

Input to NPPV

```
{ }
BEGIN
  x := A;
  y := N;
  WHILE not P(y) DO { s(x,f(y)) = f(N) }
    BEGIN
      x := s(x,y);
      y := r(y)
    END;
  x := s(x,h(y))
END
{ x = f(N) }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

TRUE
     ==>
          s(A,f(N))=f(N)
--------- Remains to prove  ---------
s(A,f(N))=f(N)
=====================================

=====================================
=== Verification Condition No.: 2 ===

s(x,f(y))=f(N)
     ==>
          s(x,f(y))=f(N)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

s(x,f(y))=f(N) AND NOT (NOT P(y))
     ==>
          s(x,h(y))=f(N)
--------- Remains to prove  ---------
s(x,f(y))=f(N) AND P(y)
     ==>
          s(x,h(y))=f(N)
=====================================

=====================================
=== Verification Condition No.: 4 ===

s(x,f(y))=f(N) AND NOT P(y)
     ==>
          s(s(x,y),f(r(y)))=f(N)
--------- Remains to prove  ---------
s(x,f(y))=f(N) AND NOT P(y)
     ==>
          s(s(x,y),f(r(y)))=f(N)
=====================================
```

FPP: in this abstract example it is not possible to give a termination function for the WHILE-loop. Since FPP requires such a termination function the example is not appropriate for FPP.

4. factfor

The factorial function is computed using a FOR-loop.

Input to NPPV

```
{ N >= 0}
BEGIN
 prod := 1;              { prod = 1}
 FOR i := 1 TO N DO
            { prod = fact(i-1) }
     prod := prod * i
END
{ prod = fact(N) }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

N>=0
     ==>
           1=1
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

prod=1
     ==>
           prod=fact(1-1)
--------- Remains to prove  ---------
1=fact(0)
=====================================

=====================================
=== Verification Condition No.: 3 ===

prod=fact(N+1-1)
     ==>
           prod=fact(N)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

prod=1 AND N<1
     ==>
           prod=fact(N)
--------- Remains to prove  ---------
N<1
     ==>
           1=fact(N)
=====================================

=====================================
=== Verification Condition No.: 5 ===

prod=fact(i-1) AND 1<=i AND i<=N
     ==>
           prod*i=fact(i+1-1)
--------- Remains to prove  ---------
1<=i AND i<=N
     ==>
           i*fact(i-1)=fact(i)
=====================================
```

Input to FPP:

In FPP a FOR-loop requires a precondition, a postcondition and an invariant. In FPP, as in NPPV, the loop variable occurs in the invariant.

```
   --!pre: n >= 0 and n = n_i;
prod := 1;
   --!pre : prod = 1 and n >= 0 and n = n_i;
   --!post: prod = factorial(n) and n = n_i;
   --!inv : prod = factorial(i) and n = n_i;
FOR i IN 1 .. n LOOP
    prod := prod * i;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.02, 8:29
The answer to your query is:

--!pre       : (n >= 0 AND n = n_i)
--> wp       : (n >= 0 AND n = n_i)
--> vc       : (True)
--> Result: proved
prod := 1;

--!pre       : (prod = 1 AND n >= 0 AND n = n_i)
--!post      : (prod = Factorial(n) AND n = n_i)
--!inv       : (prod = Factorial(i) AND n = n_i)
-->functionality ---------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :(1 <= n AND prod = 1 AND n >= 0 AND n = n_i ==> prod = 1 AND n = n_i)
--> Result    : proved
-->induction :     (1 <= n AND prod = Factorial(-1 + i) AND n = n_i)
-->           ==> (i*prod = Factorial(i) AND n = n_i)
--> Result    : proved
-->final     :     (1 <= n AND prod = Factorial(n) AND n = n_i)
-->           ==> (prod = Factorial(n) AND n = n_i)
--> Result    : proved
-->null loop :     (1 >= 1 + n AND prod = 1 AND n >= 0 AND n = n_i)
-->           ==> (prod = Factorial(n) AND n = n_i)
--> Result    : proved
FOR i IN 1 .. n  LOOP
   prod := prod * i;
END LOOP;
```

5. factforty

The same example as the last one (factfor) but with assertions which express the type constraints applicable.

In factfor both provers assume integer = Z, i.e. they do not take into account that on the computer we have limited ranges for the integer types [NW 89; KW 97].

Input to NPPV

```
{ 0 <= N  and N <= 7}
BEGIN
 prod := 1;              { prod = 1 and 0 <= N and N <= 7}
 FOR i := 1 TO N DO      { prod = fact(i-1) and 0 <= i and i-1 <= 7 and N <= 7 }
     prod  := prod * i
END
{ prod = fact(N) and prod <= 32767}
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

0<=N AND N<=7
     ==>
           1=1 AND 0<=N AND N<=7
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

prod=1 AND 0<=N AND N<=7
     ==>
           prod=fact(1-1) AND 0<=1 AND 1-1<=7 AND N<=7
--------- Remains to prove  ---------
0<=N AND N<=7
     ==>
           1=fact(0)
=====================================

=====================================
=== Verification Condition No.: 3 ===

prod=fact(N+1-1) AND 0<=N+1 AND N+1-1<=7 AND N<=7
     ==>
           prod=fact(N) AND prod<=32767
--------- Remains to prove  ---------
0<=N+1 AND N<=7
     ==>
           fact(N)<=32767
=====================================

=====================================
=== Verification Condition No.: 4 ===

prod=1 AND 0<=N AND N<=7 AND N<1
     ==>
           prod=fact(N) AND prod<=32767
--------- Remains to prove  ---------
0<=N AND N<=7 AND N<1
     ==>
           1=fact(N)
=====================================

=====================================
=== Verification Condition No.: 5 ===

prod=fact(i-1) AND 0<=i AND i-1<=7 AND N<=7 AND 1<=i AND i<=N
     ==>
           prod*i=fact(i+1-1) AND 0<=i+1 AND i+1-1<=7 AND N<=7
--------- Remains to prove  ---------
0<=i AND i<=8 AND N<=7 AND 1<=i AND i<=N
     ==>
           i*fact(i-1)=fact(i) AND 0<=i+1 AND i<=7
=====================================
```

Input to FPP

```
   --!pre: n >= 0 and n <= 7 and n = n_i;
prod := 1;
   --!pre :prod = 1 and n >= 0 and n <= 7 and n = n_i;
   --!post: prod = factorial(n) and prod <= 32767 and n = n_i;
   --!inv : prod = factorial(i) and 1 <= i+1 and n <= 7 and n = n_i;
FOR i IN 1 .. n LOOP
    prod := prod * i;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.01, 16:00
The answer to your query is:

--!pre       : (n >= 0 AND n = n_i)
--> wp       : (n >= 0 AND n = n_i)
--> vc       : (True)
--> Result: proved
prod := 1;
```

```
--!pre       : (prod = 1 AND n >= 0 AND n = n_i)
--!post      : (prod = Factorial(n) AND n = n_i)
--!inv       : (prod = Factorial(i) AND n = n_i)
-->functionality ------------------------------------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :(1 <= n AND prod = 1 AND n >= 0 AND n = n_i ==> prod = 1 AND n = n_i)
-->Result    : proved
-->induction :     (1 <= n AND prod = Factorial(-1 + i) AND n = n_i)
-->            ==> (i*prod = Factorial(i) AND n = n_i)
-->Result    : proved
-->final     :     (1 <= n AND prod = Factorial(n) AND n = n_i)
-->            ==> (prod = Factorial(n) AND n = n_i)
-->Result    : proved
-->null loop :     (1 >= 1 + n AND prod = 1 AND n >= 0 AND n = n_i)
-->            ==> (prod = Factorial(n) AND n = n_i)
-->Result    : proved
FOR i IN 1 .. n  LOOP
   prod := prod * i;
END LOOP;
```

6. fastmul A fast multiplication algorithm.

Input to NPPV

```
{  }
  BEGIN
  x := A;
  y := B;
  s := 0;                /*   { s=0  and x = A and y = B } */
    WHILE x <> 0 DO          { x*y + s = A*B }
      BEGIN
        WHILE x mod 2 = 0 DO { x*y + s = A*B }
          BEGIN
            y := 2*y;
            x := x div 2
          END ;
        s := s + y;
        x := x - 1
      END
  END
{ s = A*B }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

TRUE
     ==>
           A*B+0=A*B
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

x*y+s=A*B
     ==>
           x*y+s=A*B
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

x*y+s=A*B AND NOT x<>0
     ==>
           s=A*B
=========  Proof succeeded  =========
=====================================
=====================================
=== Verification Condition No.: 4 ===

x*y+s=A*B AND x<>0
     ==>
           x*y+s=A*B
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 5 ===

x*y+s=A*B AND NOT x mod 2=0
     ==>
           (x-1)*y+s+y=A*B
--------- Remains to prove  ---------
x*y+s=A*B AND x mod 2<>0
     ==>
           (x-1)*y+s+y=A*B
=====================================

=====================================
=== Verification Condition No.: 6 ===

x*y+s=A*B AND x mod 2=0
     ==>
           x div 2*2*y+s=A*B
--------- Remains to prove  ---------
x*y+s=A*B AND x mod 2=0
     ==>
           x div 2*2*y+s=A*B
=====================================
```

FPP: WHILE-loops must always have a termination function. This example does not contain a termination function and is therefore not appropriate for FPP.

7. fastmult

The fastmul algorithm (previous example) with a termination function.

Input to NPPV

```
{ A >= 0 }
  BEGIN
  x := A;
  y := B;
  s := 0;
    WHILE x <> 0 DO [x] { x*y + s = A*B and x >= 0 }
      BEGIN
        WHILE x mod 2 = 0 DO [x]{ x*y + s = A*B and x > 0 }
          BEGIN
            y := 2*y;
            x := x div 2
          END ;
        s := s + y ;
        x := x - 1
      END
  END
{ s = A*B }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

A>=0
     ==>
           A*B+0=A*B AND A>=0
```

```
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

x*y+s=A*B AND x>=0
     ==>
           x*y+s=A*B AND x>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

x<>0 AND x*y+s=A*B AND x>=0
     ==>
           x>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

x*y+s=A*B AND x>=0 AND NOT x<>0
     ==>
           s=A*B
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 5 ===

x<>0 AND x*y+s=A*B AND x>=0 AND x>=0
     ==>
           x*y+s=A*B AND x>0
--------- Remains to prove  ---------
x<>0 AND x*y+s=A*B AND 0<=x
     ==>
           0<x
=====================================

=====================================
=== Verification Condition No.: 6 ===

x mod 2=0 AND x<>0 AND x*y+s=A*B AND x>=0 AND x>=0
     ==>
           x>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 7 ===

x*y+s=A*B AND x>0 AND NOT x mod 2=0
     ==>
           (x-1)*y+s+y=A*B AND x-1>=0 AND x-1>=0
--------- Remains to prove  ---------
x*y+s=A*B AND 0<x AND x mod 2<>0
     ==>
           (x-1)*y+s+y=A*B AND 1<=x
=====================================

=====================================
=== Verification Condition No.: 8 ===

x mod 2=0 AND x*y+s=A*B AND x>0 AND x>=0
     ==>
           x div 2*2*y+s=A*B AND x div 2>0 AND x div 2>=0
--------- Remains to prove  ---------
x mod 2=0 AND x*y+s=A*B AND 0<x
     ==>
           x div 2*2*y+s=A*B AND 0<x div 2
=====================================
=====================================
=== Verification Condition No.: 9 ===

x mod 2=0 AND x*y+s=A*B AND x>0 AND x=x
     ==>
           x div 2<x
--------- Remains to prove  ---------
x mod 2=0 AND x*y+s=A*B AND 0<x
     ==>
           x div 2<x
=====================================

=====================================
=== Verification Condition No.: 10 ===

x<>0 AND x*y+s=A*B AND x>=0 AND x=x
     ==>
           x*y+s=A*B AND x>0
--------- Remains to prove  ---------
x<>0 AND x*y+s=A*B AND 0<=x
     ==>
           0<x
=====================================

=====================================
=== Verification Condition No.: 11 ===

x mod 2=0 AND x<>0 AND x*y+s=A*B AND x>=0 AND x=x
     ==>
           x>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 12 ===

x*y+s=A*B AND x>0 AND NOT x mod 2=0
     ==>
           x-1<x
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 13 ===

x mod 2=0 AND x*y+s=A*B AND x>0 AND x>=0
     ==>
           x div 2*2*y+s=A*B AND x div 2>0 AND x div 2>=0
--------- Remains to prove  ---------
x mod 2=0 AND x*y+s=A*B AND 0<x
     ==>
           x div 2*2*y+s=A*B AND 0<x div 2
=====================================

=====================================
=== Verification Condition No.: 14 ===

x mod 2=0 AND x*y+s=A*B AND x>0 AND x=x
     ==>
           x div 2<x
--------- Remains to prove  ---------
x mod 2=0 AND x*y+s=A*B AND 0<x
     ==>
           x div 2<x
=====================================
```

Input to FPP

In FPP a WHILE-loop requires a precondition and a postcondition. Since in the NPPV example only an invariant is provided, we must generate the missing assertions. The precondition for the outer loop is simply the conjunction of the precondition of the program and the postcondition of the initializations. The postcondition of the outer loop is the postcondition of the program. The precondition of the inner loop is the same as the invariant of the inner loop. The postcondition of the inner loop is the conjunction of the invariant and the negation of the condition of the inner loop.

```
   -- Example 7
   --!pre : x_i>=0 and x=x_i and y=y_i and s=0;
   --!post: s = x_i * y_i;
   --!inv : x*y + s = x_i * y_i and x>=0;
   --!term: x;
WHILE x /= 0 LOOP
      --!pre : x*y + s = x_i * y_i and x>0;
      --!post: x*y + s = x_i * y_i and x>0
      --!post: and x mod 2 /= 0;
      --!inv : x*y + s = x_i * y_i and x>0;
      --!term: x;
   WHILE x mod 2 = 0 LOOP
      y := 2*y;
      x := x / 2;
   END LOOP;
   s := s + y;
   x := x - 1;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.10.12, 11:45
The answer to your query is:

--!pre       : (x_i >= 0 AND x = x_i AND y = y_i)
--!post      : (0 = x_i*y_i)
--!inv       : (x*y = x_i*y_i AND x >= 0)
--!term      : (x)
-->functionality --------------------------
-->initial   :(x_i >= 0 AND x = x_i AND y = y_i ==> x*y = x_i*y_i AND x >= 0)
--> Result    : proved
-->induction :     (x /= 0 AND 1 + x*y = x_i*y_i AND x >= 0)
-->            ==>      ((exists x_i,y_i: 2 + x*y = x_i*y_i AND x >= 1))
-->                 AND (forall(($2,$3)),
-->                              (forall((x_i,y_i)),
-->                                       (2 + x*y = x_i*y_i AND x >= 1)
-->                                   ==>     (2 + $2*$3 = x_i*y_i)
-->                                        AND ($2 >= 1) AND ($2 mod 2 /= 0))
-->                          ==> (2 + $3 + (-1 + $2)*$3 = x_i*y_i AND -1 + $2 >= 0))
--> Result    : proved
-->final     :(x = 0 AND x*y = x_i*y_i AND x >= 0 ==> 0 = x_i*y_i)
--> Result    : proved
-->termination --------------------------
-->initial   :(x /= 0 AND x*y = x_i*y_i AND x >= 0 ==> x >= 1)
--> Result    : proved
-->induction :     (x /= 0 AND 1 + x*y = x_i*y_i AND x >= 0)
-->            ==>      ((exists x_i,y_i: 2 + x*y = x_i*y_i AND x >= 1))
-->                 AND (forall(($2,$3)),
-->                              (forall((x_i,y_i)),
-->                                       (2 + x*y = x_i*y_i AND x >= 1)
-->                                   ==>     (2 + $2*$3 = x_i*y_i)
-->                                        AND ($2 >= 1) AND ($2 mod 2 /= 0))
-->                          ==> (x >= $2))
--> Result    : not proved
--> fc       :      (1 + x*y - x_i*y_i = 0) AND (-(x*y) + $03*$04 = 0)
-->             AND ($03 >= 1 + x) AND ($03 >= 1) AND (x /= 0)
-->             AND ($03 mod 2 /= 0) AND (x >= 0)

WHILE x /= 0 LOOP

   --!pre       : (x*y = x_i*y_i AND x >= 1)
   --!post      : (x*y = x_i*y_i AND x >= 1 AND x mod 2 /= 0)
   --!inv       : (x*y = x_i*y_i AND x >= 1)
   --!term      : (x)
   -->functionality --------------------------
   -->initial   :(True)
   --> Result    : proved
   -->induction :     (x mod 2 = 0 AND 1 + x*y = x_i*y_i AND x >= 1)
-->                ==> (2 + x*y = x_i*y_i AND x/2 >= 1)
   --> Result    : not proved
--> fc       : (x*y - x_i*y_i = 0 AND 1 - x/2 >= 1 AND -1 + x >= 0 AND x mod 2 = 0)

   -->final     :     (x mod 2 /= 0 AND 1 + x*y = x_i*y_i AND x >= 1)
-->                ==> (2 + x*y = x_i*y_i AND x >= 1 AND x mod 2 /= 0)
   --> Result    : proved
   -->termination ----------------------------
   -->initial   :(x mod 2 = 0 AND x*y = x_i*y_i AND x >= 1 ==> x >= 1)
   --> Result    : proved
   -->induction :(x mod 2 = 0 AND x*y = x_i*y_i AND x >= 1 ==> x >= 1 + x/2)
   --> Result    : not proved
--> fc       : (x*y - x_i*y_i = 0 AND 1 - x/2 >= 1 AND -1 + x >= 0 AND x mod 2 = 0)

   WHILE x MOD 2 = 0 LOOP
      y := 2 * y;
      x := x / 2;
   END LOOP;

   s := s + y;
   x := x - 1;
END LOOP;
```

This is the first example in which auxiliary variables are generated.
In this example we observe furthermore that the theorem prover is pushed to its limits, because it is not able to prove all verification conditions. If we look at the last induction condition we see that it holds because `x mod 2 = 0 AND x >= 1 ==> x >= 1 + x/2.` We see also easily that the FC cannot be fulfilled because of `x <= 0 AND x >= 1.`

8. fastmultty

This is the same as example 7 (fastmult) but additionally with assertions which express limitations for the ranges of the variables.

Input to NPPV

```
{ A >= 0 and -32767 <= A*B and A*B <= 32767 and -32767 <= B and B <=  32767}
  BEGIN
  x := A;
  y := B;
  s := 0;
    WHILE x <> 0 DO [x] { x*y + s = A*B and x >= 0 and -32767 <= A*B and
                A*B <= 32767}
      BEGIN
        WHILE x mod 2 = 0 DO [x]{ x*y + s = A*B and x > 0 and -32767 <=
                              A*B and A*B <= 32767}
          BEGIN
            y := 2*y;
            x := x div 2
          END ;
        s := s + y ;
        x := x - 1
      END
  END
{ s = A*B and -32767 <= s and s <= 32767}
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

A>=0 AND -32767<=A*B AND A*B<=32767 AND -32767<=B AND B<=32767
     ==>
           A*B+0=A*B AND A>=0 AND -32767<=A*B AND A*B<=32767
=========  Proof succeeded  =========
=====================================

=====================================
   === Verification Condition No.: 2 ===

   x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767
        ==>
              x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 3 ===

   x<>0 AND x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767
        ==>
              x>=0
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 4 ===

   x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767 AND NOT x<>0
        ==>
              s=A*B AND -32767<=s AND s<=32767
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 5 ===

   x<>0 AND x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767 AND x>=0
        ==>
              x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767
   --------- Remains to prove  ---------
   x<>0 AND x*y+s=A*B AND 0<=x AND -32767<=A*B AND A*B<=32767 AND 0<=x
        ==>
              0<x
   =====================================

   =====================================
   === Verification Condition No.: 6 ===

   x mod 2=0 AND x<>0 AND x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767 AND x>=0
        ==>
              x>=0
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 7 ===

   x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767 AND NOT x mod 2=0
        ==>
              (x-1)*y+s+y=A*B AND x-1>=0 AND -32767<=A*B AND A*B<=32767 AND x-1>=0
   --------- Remains to prove  ---------
   x*y+s=A*B AND 0<x AND -32767<=A*B AND A*B<=32767 AND x mod 2<>0
        ==>
              (x-1)*y+s+y=A*B AND 1<=x
   =====================================

   =====================================
   === Verification Condition No.: 8 ===

   x mod 2=0 AND x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767 AND x>=0
        ==>
              x div 2*2*y+s=A*B AND x div 2>0 AND -32767<=A*B AND A*B<=32767 AND x div
2>=0
   --------- Remains to prove  ---------
   x mod 2=0 AND x*y+s=A*B AND 0<x AND -32767<=A*B AND A*B<=32767 AND 0<=x
        ==>
              x div 2*2*y+s=A*B AND 0<x div 2 AND 0<=x div 2
   =====================================

   =====================================
   === Verification Condition No.: 9 ===
   x mod 2=0 AND x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767 AND x=x
        ==>
              x div 2<x
   --------- Remains to prove  ---------
   x mod 2=0 AND x*y+s=A*B AND 0<x AND -32767<=A*B AND A*B<=32767
        ==>
              x div 2<x
   =====================================

   =====================================
   === Verification Condition No.: 10 ===

   x<>0 AND x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767 AND x=x
        ==>
              x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767
   --------- Remains to prove  ---------
   x<>0 AND x*y+s=A*B AND 0<=x AND -32767<=A*B AND A*B<=32767
        ==>
              0<x
   =====================================

   =====================================
   === Verification Condition No.: 11 ===

   x mod 2=0 AND x<>0 AND x*y+s=A*B AND x>=0 AND -32767<=A*B AND A*B<=32767 AND x=x
        ==>
              x>=0
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 12 ===

   x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767 AND NOT x mod 2=0
        ==>
              x-1<x
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 13 ===

   x mod 2=0 AND x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767 AND x>=0
        ==>
              x div 2*2*y+s=A*B AND x div 2>0 AND -32767<=A*B AND A*B<=32767 AND x div
2>=0
   --------- Remains to prove  ---------
   x mod 2=0 AND x*y+s=A*B AND 0<x AND -32767<=A*B AND A*B<=32767 AND 0<=x
        ==>
              x div 2*2*y+s=A*B AND 0<x div 2 AND 0<=x div 2
   =====================================

   =====================================
   === Verification Condition No.: 14 ===

   x mod 2=0 AND x*y+s=A*B AND x>0 AND -32767<=A*B AND A*B<=32767 AND x=x
        ==>
              x div 2<x
   --------- Remains to prove  ---------
   x mod 2=0 AND x*y+s=A*B AND 0<x AND -32767<=A*B AND A*B<=32767
        ==>
              x div 2<x
   =====================================
```

Input to FPP

```
   s := 0;
      --!pre : s=0 and x=x_i and y=y_i and x_i>=0 and -32767<=y_i and y_i<=32767 and
      --!pre : -32767<=x_i*y_i and x_i*y_i<=32767;
      --!post: s = x_i*y_i and -32767<=s and s<=32767;
      --!inv : x*y + s = x_i*y_i and x>=0 and -32767<=x_i*y_i and x_i*y_i<=32767;
      --!term: x;
   WHILE x /= 0 LOOP
         --!pre : x*y + s = x_i*y_i and x>0 and -32767<=x_i*y_i and x_i*y_i<=32767;
         --!post: x*y + s = x_i* y_i and x>0 and -32767<=x_i*y_i and x_i*y_i<=32767
         --!post: and x mod 2 /= 0;
         --!inv : x*y + s = x_i*y_i and x>0 and -32767<=x_i*y_i and x_i*y_i<=32767;
         --!term: x;
      WHILE x mod 2 = 0 LOOP
         y := 2*y;
         x := x / 2;
      END LOOP;
      s := s + y;
      x := x - 1;
   END LOOP;
```

Output from FPP

As in example 7 we observe that the theorem prover is pushed to its limits, because it is not able to prove all verification conditions. With the preceding input it did not give an answer within 24 h. For an input with smaller ranges it gave an answer which shows that it cannot prove all VCs.

9. fibo Iterative computation of the Nth Fibonacci number.

Input to NPPV

```
   { N >= 1 }
   BEGIN
         previous := 0;
         current  := 1;
         count    := 1;
         WHILE count < N  DO      { count >= 1 and count  <= N  and
                                    previous = fibo(count-1) and
                                    current  = fibo(count)   }
           BEGIN
             x := current;
             current := current + previous;
             previous := x;
             count := count+1
           END
   END
   { current = fibo(N) }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

N>=1
       ==>
             1>=1 AND 1<=N AND 0=fibo(1-1) AND 1=fibo(1)
--------- Remains to prove  ---------
1<=N
       ==>
             0=fibo(0) AND 1=fibo(1)
=====================================

=====================================
=== Verification Condition No.: 2 ===

count>=1 AND count<=N AND previous=fibo(count-1) AND current=fibo(count)
       ==>
             count>=1 AND count<=N AND previous=fibo(count-1) AND current=fibo(count)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===
count>=1 AND count<=N AND previous=fibo(count-1) AND
current=fibo(count) AND NOT count<N
       ==>
             current=fibo(N)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

count>=1 AND count<=N AND previous=fibo(count-1) AND current=fibo(count) AND count<N
       ==>
             count+1>=1   AND   count+1<=N   AND   current=fibo(count+1-1)   AND   cur-
rent+previous=fibo(count+1)
--------- Remains to prove  ---------
1<=count AND count<N
       ==>
             1<=count+1 AND count+1<=N AND fibo(count)+fibo(count-1)=fibo(count+1)
=====================================
```

FPP: WHILE-loops must always have a termination function. This example does not contain a termination function and is therefore not appropriate for FPP.

10. fibot

The same as example 9 but with a termination function.

Input to NPPV

```
  { N >= 1 and N <= 23}
  BEGIN
        previous := 0;
        current  := 1;
        count    := 1;
        WHILE count < N  DO      [N-count] { count >= 1 and count <= N  and
                                   previous = fibo(count-1) and
                                   current  = fibo(count) and fibo(N) <= 32767  }
          BEGIN
            x := current;
            current := current + previous;
            previous := x;
            count := count+1
          END
  END
  { current = fibo(N) and current <= 32767 }
¦          previous := x;
          count := count+1
        END
END
{ current = fibo(N) }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

N>=1 AND N<=23
         ==>
                1>=1 AND 1<=N AND 0=fibo(1-1) AND 1=f
ibo(1) AND fibo(N)<=32767

--------- Remains to prove  ---------
1<=N AND N<=23
         ==>
                0=fibo(0) AND 1=fibo(1) AND fibo(N)<=
32767
=====================================
=====================================
=== Verification Condition No.: 2 ===

count>=1 AND count<=N AND previous=fibo(count-1) AND
current=fibo(count) AND fibo(N)<=32767
         ==>
                count>=1 AND count<=N AND previous=fi
bo(count-1) AND current=fibo(count) AND fibo(N)<=3276
7

=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

count<N AND count>=1 AND count<=N AND previous=fibo(c
ount-1) AND current=fibo(count) AND fibo(N)<=32767
         ==>
                N-count>=0

=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

count>=1 AND count<=N AND previous=fibo(count-1) AND
current=fibo(count) AND fibo(N)<=32767 AND NOT count<
N
         ==>
                current=fibo(N) AND current<=32767

--------- Remains to prove  ---------
1<=count AND count<=N AND fibo(N)<=32767 AND N<=count

         ==>
                fibo(count)=fibo(N) AND fibo(count)<=
32767
=====================================

=====================================
=== Verification Condition No.: 5 ===

count<N AND count>=1 AND count<=N AND previous=fibo(c
ount-1) AND current=fibo(count) AND N-count>=0
         ==>
                count+1>=1 AND count+1<=N AND current
=fibo(count+1-1) AND current+previous=fibo(count+1) A
ND N-(count+1)>=0

--------- Remains to prove  ---------
count<N AND 1<=count AND count<=N
         ==>
                1<=count+1 AND count+1<=N AND fibo(co
unt)+fibo(count-1)=fibo(count+1) AND count+1<=N
=====================================

=====================================
=== Verification Condition No.: 6 ===

count<N AND count>=1 AND count<=N AND previous=fibo(c
ount-1) AND current=fibo(count) AND fibo(N)<=32767 AN
D N-count=N-count
         ==>
                N-(count+1)<N-count

=========  Proof succeeded  =========
=====================================
```

Input to FPP

In FPP a WHILE-loop requires a precondition and a postcondition. Since in the NPPV example only an invariant is provided, we must provide the missing assertions. The precondition for the loop is simply the precondition of the program conjunctively connected with the postcondition of the initializations. The postcondition of the loop is the postcondition of the program.

```
   --!pre: n >= 1 and n <= 23 and n = n_i;
previous := 0;
current  := 1;
count    := 1;
   --!pre : n >= 1 and n <= 23 and n = n_i and previous = 0 and current = 1
   --!pre: and count = 1;
   --!post: current = fib(n) and n = n_i;
   --!inv : count >= 1 and count  <= n  and previous = fib(count-1) and
   --!inv : current  = fib(count) and n = n_i;
   --!term: n-count;
WHILE count < n  LOOP
   x := current;
   current := current + previous;
   previous := x;
   count := count+1;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.02, 9:31
The answer to your query is:

--!pre       : (n >= 1 AND n <= 23 AND n = n_i)
--> wp       : (n >= 1 AND n <= 23 AND n = n_i)
--> vc       : (True)
--> Result: proved
previous := 0;
current := 1;
count := 1;

--!pre       : (n >= 1) AND (n <= 23) AND (n = n_i) AND (previous = 0)
-->            AND (current = 1) AND (count = 1)
--!post      :     (current = (-(1 - Sqrt(5))**n + (1 + Sqrt(5))**n)/(2**n*Sqrt(5)))
-->            AND (n = n_i)
--!inv       : (count >= 1) AND (count <= n) AND (previous) = Fib((-1 + count))
-->            AND (current) = Fib((count)) AND (n = n_i)
--!term      : (-count + n)
-->functionality --------------------------
-->initial   :     (n >= 1) AND (n <= 23) AND (n = n_i) AND (previous = 0)
-->                AND (current = 1) AND (count = 1)
-->            ==> (count >= 1) AND (count <= n) AND (previous) = Fib((-1 + count))
-->                AND (current) = Fib((count)) AND (n = n_i)
-->Result    : proved
-->induction :      (n >= 1 + count) AND (count >= 1) AND (count <= n)
-->                 AND (previous) = Fib((-1 + count))
-->                 AND (current) = Fib((count)) AND (n = n_i)
-->            ==>  (1 + count >= 1) AND (1 + count <= n)
-->                 AND (current) = Fib((count))
-->                 AND (current + previous) = Fib((1 + count)) AND (n = n_i)
-->Result    : proved
-->final     :     (n <= count) AND (count >= 1) AND (count <= n)
-->                AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
-->                AND (n = n_i)
-->           ==>  (current) = ((-(1 - Sqrt(5))**n + (1 + Sqrt(5))**n)/(2**n*Sqrt(5)))
-->                AND (n = n_i)
-->Result    : proved
-->termination --------------------------
-->initial   :          (n >= 1 + count) AND (count >= 1) AND (count <= n)
-->                 AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
-->                 AND (n = n_i)
-->            ==> (-count + n >= 1)
-->Result    : proved
```

```
-->induction :      (n >= 1 + count) AND (count >= 1) AND (count <= n)
-->                 AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
-->                 AND (n = n_i)
-->            ==> (-count + n >= -count + n)
-->Result    : proved
WHILE count < n LOOP
   x := current;
   current := current + previous;
   previous := x;
   count := count + 1;
END LOOP;
```

11. fibotty

The same as example 10 but with type checking assertions.

Input to NPPV

```
   { N >= 1 and N <= 23}
   BEGIN
         previous := 0;
         current  := 1;
         count    := 1;
         WHILE count < N  DO      [N-count] { count >= 1 and count <= N  and
                                    previous = fibo(count-1) and
                                    current  = fibo(count) and fibo(N) <= 32767  }
           BEGIN
             x := current;
             current := current + previous;
             previous := x;
             count := count+1
           END
   END
   { current = fibo(N) and current <= 32767 }
```

Output from NPPV

```
   =====================================
   === Verification Condition No.: 1 ===

   N>=1 AND N<=23
        ==>
              1>=1 AND 1<=N AND 0=fibo(1-1) AND 1=fibo(1) AND fibo(N)<=32767
   --------- Remains to prove  ---------
   1<=N AND N<=23
        ==>
              0=fibo(0) AND 1=fibo(1) AND fibo(N)<=32767
   =====================================

   =====================================
   === Verification Condition No.: 2 ===

   count>=1 AND count<=N AND previous=fibo(count-1) AND current=fibo(count) AND
fibo(N)<=32767
        ==>
              count>=1 AND count<=N AND previous=fibo(count-1) AND current=fibo(count)
AND fibo(N)<=32767
   =========  Proof succeeded  =========
   =====================================

   =====================================
   === Verification Condition No.: 3 ===

   count<N AND count>=1 AND count<=N AND previous=fibo(count-1) AND
current=fibo(count) AND fibo(N)<=32767
        ==>
              N-count>=0
   =========  Proof succeeded  =========
   =====================================
   =====================================
   === Verification Condition No.: 4 ===

   count>=1 AND count<=N AND previous=fibo(count-1) AND current=fibo(count) AND
fibo(N)<=32767 AND NOT count<N
        ==>
             current=fibo(N) AND current<=32767
   --------- Remains to prove  ---------
   1<=count AND count<=N AND fibo(N)<=32767 AND N<=count
        ==>
             fibo(count)=fibo(N) AND fibo(count)<=32767
   =====================================

   =====================================
   === Verification Condition No.: 5 ===

   count<N AND count>=1 AND count<=N AND previous=fibo(count-1) AND
current=fibo(count) AND fibo(N)<=32767 AND N-count>=0
        ==>
             count+1>=1 AND count+1<=N AND current=fibo(count+1-1) AND cur-
rent+previous=fibo(count+1) AND fibo(N)<=32767 AND N-(count+1)>=0
   --------- Remains to prove  ---------
   count<N AND 1<=count AND count<=N AND fibo(N)<=32767 AND count<=N
        ==>
             1<=count+1 AND count+1<=N AND fibo(count)+fibo(count-1)=fibo(count+1) AND
count+1<=N
   =====================================

   =====================================
   === Verification Condition No.: 6 ===

   count<N AND count>=1 AND count<=N AND previous=fibo(count-1) AND
current=fibo(count) AND fibo(N)<=32767 AND N-count=N-count
        ==>
             N-(count+1)<N-count
   =========  Proof succeeded  =========
   =====================================
```

Input to FPP

```
   --!pre: n >= 1 and n <= 23 and n = n_i;
      previous := 0;
      current  := 1;
      count    := 1;
   --!pre : n >= 1 and n <= 23 and n = n_i and previous = 0 and current = 1
   --!pre: and count = 1;
   --!post: current = fib(n) and current <= 32767 and n = n_i;
   --!inv : count >= 1 and count  <= n  and previous = fib(count-1) and
   --!inv : current = fib(count) and fib(n) <= 32767 and n = n_i;
   --!term: n-count;
      WHILE count < n  LOOP
            x := current;
            current := current + previous;
            previous := x;
            count := count+1;
      END LOOP;
```

Output from FPP

```
   FPP (Frege Program Prover) University of Jena, Germany
   User: 141.35.14.241       At: 1999.09.23, 13:05
   The answer to your query is:

   --!pre       : (n >= 1 AND n <= 23 AND n = n_i)
   --> wp       : (n >= 1 AND n <= 23 AND n = n_i)
   --> vc       : (True)
   --> Result: proved
   previous := 0;
   current := 1;
   count := 1;
   --!pre       :      (n >= 1) AND (n <= 23) AND (n = n_i) AND (previous = 0)
   -->            AND (current = 1) AND (count = 1)
   --!post      :      (current = (-(1-Sqrt(5))**n + (1 + Sqrt(5))**n)/(2**n*Sqrt(5)))
   -->            AND (current <= 32767) AND (n = n_i)
   --!inv       :      (count >= 1) AND (count <= n) AND (previous) = Fib((-1 +
count))           AND    (current) = Fib((count))
   -->            AND ((-(1 - Sqrt(5))**n + (1 + Sqrt(5))**n)/(2**n*Sqrt(5)) <= 32767)
   -->            AND (n = n_i)
   --!term      : (-count + n)
   -->functionality ---------------------------
   -->initial   :        (n >= 1) AND (n <= 23) AND (n = n_i) AND (previous = 0)
   -->               AND (current = 1) AND (count = 1)
   -->          ==>      (count >= 1) AND (count <= n)
   -->               AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
   -->               AND ((-(1-Sqrt(5))**n + (1+Sqrt(5))**n)/(2**n*Sqrt(5))) <=
(32767)              AND (n = n_i)
   --> Result    : not proved
   --> fc       :    (count = current) AND (-32767) + (2**(-n_i)) * (1/Sqrt(5)) *
   -->               (-(-Sqrt(5) + current)**n_i + (Sqrt(5) + current)**n_i) > (0)
   -->                 AND (23-n_i >= 0) AND (-current + n_i >= 0) AND (current = 1)
   -->induction :        (n >= 1 + count) AND (count >= 1) AND (count <= n)
   -->               AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
   -->               AND ((-(1-Sqrt(5))**n + (1+Sqrt(5))**n)/(2**n*Sqrt(5))) <=
(32767)              AND (n = n_i)
   -->          ==>      (1+count >= 1) AND (1+count <= n) AND (current) =
Fib((count))         AND (current + previous) = Fib((1 + count))
   -->               AND ((-(1-Sqrt(5))**n + (1+Sqrt(5))**n)/(2**n*Sqrt(5))) <=
(32767)              AND (n = n_i)
   --> Result    : proved
   -->final     :        (n <= count) AND (count >= 1) AND (count <= n)
   -->               AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
   -->               AND ((-(1-Sqrt(5))**n + (1+Sqrt(5))**n)/(2**n*Sqrt(5))) <=
(32767)              AND (n = n_i)
   -->          ==>      (current) = ((-(1-Sqrt(5))**n+
(1+Sqrt(5))**n)/(2**n*Sqrt(5)))AND (current <= 32767) AND (n = n_i)
   --> Result    : proved
   -->termination --------------------------
   -->initial   :        (n >= 1 + count) AND (count >= 1) AND (count <= n)
   -->               AND (previous) = Fib((-1 + count)) AND (current) = Fib((count))
   -->               AND ((-(1-Sqrt(5))**n + (1+Sqrt(5))**n)/(2**n*Sqrt(5))) <=
(32767)              AND (n = n_i)
   -->          ==>      (-count + n >= 1)
   --> Result    : proved
   -->induction :        (n >= 1 + count) AND (count >= 1) AND (count <= n)
   -->               AND (previous) = Fib((-1 + count)) AND (current)= Fib((count))
   -->               AND ((-(1-Sqrt(5))**n + (1+Sqrt(5))**n)/(2**n*Sqrt(5))) <=
(32767)              AND (n = n_i)
   -->          ==> (-count + n >= -count + n)
   --> Result    : proved
   WHILE count < n LOOP
      x := current;
      current := current + previous;
      previous := x;
      count := count + 1;
   END LOOP;
```

12. gauss   Summing up the first n non negative integers.

Input to NPPV

```
   { N > 0 }
   BEGIN
     x := 0;
     sum := 0;
     WHILE x < N DO  { sum=x*(x+1) div 2 and x <= N}
       BEGIN
         x := x+1 ;
         sum := sum + x
       END
```

```
END
{ sum = N*(N+1) div 2 }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

N>0
     ==>
           0=0*(0+1) div 2 AND 0<=N
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

sum=x*(x+1) div 2 AND x<=N
     ==>
           sum=x*(x+1) div 2 AND x<=N
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

sum=x*(x+1) div 2 AND x<=N AND NOT x<N
     ==>
           sum=N*(N+1) div 2
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

sum=x*(x+1) div 2 AND x<=N AND x<N
     ==>
           sum+x+1=(x+1)*(x+1+1) div 2 AND x+1<=N
--------- Remains to prove  ---------
x<N
     ==>
           x+1<=N
=====================================
```

FPP: WHILE-loops must always have a termination function. This example does not contain a termination function and is therefore not appropriate for FPP.

13. gausst The same as example 12 but with a termination function.

Input to NPPV

```
{ N > 0 }
BEGIN
  x := 0;
  sum := 0;
  WHILE x < N DO  [ N-x ]   { sum = x*(x+1)/2 and x <= N  }
    BEGIN
      x := x+1;
      sum := sum + x
    END
END
{ sum = N*(N+1)/2 }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

N>0
     ==>
           0=0*(0+1) div 2 AND 0<=N
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

sum=x*(x+1) div 2 AND x<=N
     ==>
           sum=x*(x+1) div 2 AND x<=N
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

x<N AND sum=x*(x+1) div 2 AND x<=N
     ==>
           N-x>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

sum=x*(x+1) div 2 AND x<=N AND NOT x<N
     ==>
           sum=N*(N+1) div 2
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 5 ===

x<N AND sum=x*(x+1) div 2 AND x<=N AND N-x>=0
     ==>
           sum+x+1=(x+1)*(x+1+1) div 2 AND x+1<=N AND N-(x+1)>=0
--------- Remains to prove  ---------
x<N
     ==>
           x+1<=N
=====================================

=====================================
=== Verification Condition No.: 6 ===

x<N AND sum=x*(x+1) div 2 AND x<=N AND N-x=N-x
     ==>
           N-(x+1)<N-x
=========  Proof succeeded  =========
=====================================
```

Input to FPP

In FPP the identifier `sum` is reserved for the 4-place function `sum(e,v,l,u)`, where v is a variable and e, l and u are expressions. The meaning of `sum(e,v,l,u)` is the sum of all values of e, where v runs through the values from l to u. `sum(e,v,l,u)` is only allowed in assertions.

```
--  Example 13
--!pre: 0 < n and n = n_i;
   x := 0;
   summe := 0;
--!pre : n > 0 and x = 0 and summe = 0 and n = n_i;
--!post: summe = n*(n+1)/2 and n = n_i;
--!inv : summe = x*(x+1)/2 and x <= n  and 0 < n and n = n_i;
```

```
--!term: n-x;
   WHILE x < n LOOP
      x := x+1;
      summe := summe + x;
   END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.23, 14:03
The answer to your query is:

--!pre       : (n >= 1 AND n = n_i)
--> wp       : (n >= 1 AND n = n_i)
--> vc       : (True)
--> Result: proved
x := 0;
summe := 0;

--!pre       : (n >= 1 AND x = 0 AND summe = 0 AND n = n_i)
--!post      : (summe = n*(1 + n)/2 AND n = n_i)
--!inv       : (summe = x*(1 + x)/2 AND x <= n AND n >= 1 AND n = n_i)
--!term      : (n - x)
-->functionality --------------------------
-->initial   :     (n >= 1 AND x = 0 AND summe = 0 AND n = n_i)
-->            ==> (summe = x*(1 + x)/2 AND x <= n AND n >= 1 AND n = n_i)
--> Result    : proved
-->induction :          (n >= 1 + x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n = n_i)
-->            ==>      (1 + summe + x = (1 + x)*(2 + x)/2) AND (1 + x <= n)
-->                 AND (n >= 1) AND (n = n_i)
--> Result    : proved
-->final     :   (n <= x AND summe = x*(1 + x)/2 AND x <= n AND n >= 1 AND n = n_i)
-->          ==> (summe = n*(1 + n)/2 AND n = n_i)
--> Result    : proved
-->termination --------------------------
-->initial   :          (n >= 1 + x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n = n_i)
-->            ==> (n - x >= 1)
--> Result    : proved
-->induction :          (n >= 1 + x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n = n_i)
-->            ==> (n - x >= n - x)
--> Result    : proved
WHILE x < n LOOP
   x := x + 1;
   summe := summe + x;
END LOOP;
```

14. gausstty The same as example 13 but with type checking assertions.

Input to NPPV

```
{ 0 < N and N <= 10}
BEGIN
  x := 0;
  sum := 0;
  WHILE x < N DO  [ N-x ]   { sum = x*(x+1)/2 and x <= N  and 0 < N and N <= 10 }
    BEGIN
      x := x+1;
      sum := sum + x
    END
END
{ sum = N*(N+1)/2 and sum <= 60}
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

0<N AND N<=10
     ==>
          0=0*(0+1) div 2 AND 0<=N AND 0<N AND N<=10
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

sum=x*(x+1) div 2 AND x<=N AND 0<N AND N<=10
     ==>
          sum=x*(x+1) div 2 AND x<=N AND 0<N AND N<=10
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

x<N AND sum=x*(x+1) div 2 AND x<=N AND 0<N AND N<=10
     ==>
          N-x>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

sum=x*(x+1) div 2 AND x<=N AND 0<N AND N<=10 AND NOT x<N
     ==>
          sum=N*(N+1) div 2 AND sum<=60
--------- Remains to prove  ---------
x<=N AND 0<N AND N<=10 AND N<=x
     ==>
          (x*x+x) div 2=(N*N+N) div 2 AND (x*x+x) div 2<=60
=====================================

=====================================
=== Verification Condition No.: 5 ===

x<N AND sum=x*(x+1) div 2 AND x<=N AND 0<N AND N<=10 AND N-x>=0
     ==>
          sum+x+1=(x+1)*(x+1+1) div 2 AND x+1<=N AND 0<N AND N<=10 AND N-(x+1)>=0
--------- Remains to prove  ---------
x<N AND 0<N AND N<=10 AND x<=N
     ==>
          x+1<=N
=====================================

=====================================
=== Verification Condition No.: 6 ===

x<N AND sum=x*(x+1) div 2 AND x<=N AND 0<N AND N<=10 AND N-x=N-x
     ==>
          N-(x+1)<N-x
=========  Proof succeeded  =========
=====================================
```

Input to FPP

```
--  Example 14
--!pre: 0 < n and n <= 10 and n_i;
  x := 0;
  summe := 0;
--!pre : n > 0 and n <= 10 and x = 0 and summe = 0 and n_i;
--!post: summe = n*(n+1)/2 and summe <= 60 and n_i;
--!inv : summe = x*(x+1)/2 and x <= n  and 0 < n and n <= 10 and n_i;
--!term: n-x;
```

```
    WHILE x < n LOOP
        x := x+1;
        summe := summe + x;
    END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 8:11
The answer to your query is:

--!pre       : (n >= 1 AND n <= 10 AND n_i)
--> wp       : (n >= 1 AND n <= 10 AND n_i)
--> vc       : (True)
--> Result: proved
x := 0;
summe := 0;

--!pre       : (n >= 1 AND n <= 10 AND x = 0 AND summe = 0 AND n_i)
--!post      : (summe = n*(1 + n)/2 AND summe <= 60 AND n_i)
--!inv       : (summe = x*(1 + x)/2 AND x <= n AND n >= 1 AND n <= 10 AND n_i)
--!term      : (n - x)
-->functionality --------------------------
-->initial   :     (n >= 1 AND n <= 10 AND x = 0 AND summe = 0 AND n_i)
-->            ==> (summe = x*(1 + x)/2 AND x <= n AND n >= 1 AND n <= 10 AND n_i)
--> Result    : proved
-->induction :          (n >= 1 + x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n <= 10) AND (n_i)
-->            ==>      (1 + summe + x = (1 + x)*(2 + x)/2) AND (1 + x <= n)
-->                 AND (n >= 1) AND (n <= 10) AND (n_i)
--> Result    : proved
-->final     :          (n <= x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n <= 10) AND (n_i)
-->            ==> (summe = n*(1 + n)/2 AND summe <= 60 AND n_i)
--> Result    : proved
-->termination --------------------------
-->initial   :          (n >= 1 + x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n <= 10) AND (n_i)
-->            ==> (n - x >= 1)
--> Result    : proved
-->induction :          (n >= 1 + x) AND (summe = x*(1 + x)/2) AND (x <= n)
-->                 AND (n >= 1) AND (n <= 10) AND (n_i)
-->            ==> (n - x >= n - x)
--> Result    : proved
WHILE x < n LOOP
   x := x + 1;
   summe := summe + x;
END LOOP;
```

15. linrek

Iterative solution of a linear recursive function f(x) = if P(x) then g(x) else h(f(r(x)),x),using a stack [Gum 99a].

Input to NPPV

```
{ x = A }
BEGIN
  s := Empty ;
  WHILE not P(x) do { p(x,s) = f(A) }
    BEGIN
      s := push(x,s) ;
      x := r(x)
  END;
  z := g(x) ;
  WHILE s <> Empty do { p(z,s) = f(A) }
    BEGIN
      z := h(z,top(s)) ;
      s := pop(s)
    END
```

```
END
{ z = f(A) }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

x=A
     ==>
           p(x,Empty)=f(A)
--------- Remains to prove  ---------
p(A,Empty)=f(A)
=====================================

=====================================
=== Verification Condition No.: 2 ===

p(x,s)=f(A)
     ==>
           p(x,s)=f(A)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

p(x,s)=f(A) AND NOT (NOT P(x))
     ==>
           p(g(x),s)=f(A)
--------- Remains to prove  ---------
p(x,s)=f(A) AND P(x)
     ==>
           p(g(x),s)=f(A)
=====================================

=====================================
=== Verification Condition No.: 4 ===

p(x,s)=f(A) AND NOT P(x)
     ==>
           p(r(x),push(x,s))=f(A)
--------- Remains to prove  ---------
p(x,s)=f(A) AND NOT P(x)
     ==>
           p(r(x),push(x,s))=f(A)
=====================================

=====================================
=== Verification Condition No.: 5 ===

p(z,s)=f(A)
     ==>
           p(z,s)=f(A)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 6 ===

p(z,s)=f(A) AND NOT s<>Empty
     ==>
           z=f(A)
--------- Remains to prove  ---------
p(z,Empty)=f(A)
     ==>
           z=f(A)
=====================================

=====================================
=== Verification Condition No.: 7 ===

p(z,s)=f(A) AND s<>Empty
     ==>
           p(h(z,top(s)),pop(s))=f(A)
--------- Remains to prove  ---------
```

```
p(z,s)=f(A) AND s<>Empty
     ==>
           p(h(z,top(s)),pop(s))=f(A)
=====================================
```

FPP: in this abstract example it is not possible to give a termination function for the WHILE-loop. Since FPP requires such a termination function the example is not appropriate for FPP.

16. linsearch

Computation of the index of the first occurrence of the value of x in the array b.

NPPV: quantifiers are not supported in NPPV, and therefore this example is not appropriate for NPPV.

Input to FPP

```
-- Example 16
--!pre  :     ind=1 and len>=1 and (exists j: 1<=j and j<=len: b(j) = x)
--!pre  : and x = x_i and len = len_i;
--!post :     1<=ind and ind<=len and b(ind) = x
--!post : and not((exists j: 1<=j and j<=ind-1:b(j)=x))
--!post : and x = x_i and len = len_i;
--!inv  :     1<=ind and ind <=len and not((exists j:1<=j and j<=ind-1:b(j) = x))
--!inv  : and (exists j:1<=j and j<=len:b(j) = x) and x = x_i and len = len_i;
--!term : len+1-ind;
WHILE b(ind) /= x LOOP
        ind := ind+1;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 8:48
The answer to your query is:

--!pre       :      (ind = 1) AND (len >= 1)
-->               AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->               AND (x = x_i) AND (len = len_i)
--!post      :      (1 <= ind) AND (ind <= len) AND (b(ind) = x)
-->               AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->               AND (x = x_i) AND (len = len_i)
--!inv       :      (1 <= ind) AND (ind <= len)
-->               AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->               AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->               AND (x = x_i) AND (len = len_i)
--!term      : (1 - ind + len)
-->functionality --------------------------
-->initial   :          (ind = 1) AND (len >= 1)
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
-->             ==>     (1 <= ind) AND (ind <= len)
-->                 AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
--> Result    : proved
-->induction :          (b(ind) /= x) AND (1 <= ind) AND (ind <= len)
-->                 AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
-->             ==>     (1 <= 1 + ind) AND (1 + ind <= len)
-->                 AND (Not(exists j: 1 <= j AND j <= ind AND b(j) = x))
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
--> Result    : proved
```

```
-->final     :          (b(ind) = x) AND (1 <= ind) AND (ind <= len)
-->                 AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
-->            ==>      (1 <= ind) AND (ind <= len) AND (b(ind) = x)
-->                 AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
--> Result    : proved
-->termination --------------------------
-->initial   :          (b(ind) /= x) AND (1 <= ind) AND (ind <= len)
-->                 AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
-->            ==> (1 - ind + len >= 1)
--> Result    : proved
-->induction :          (b(ind) /= x) AND (1 <= ind) AND (ind <= len)
-->                 AND (Not(exists j: 1 <= j AND j <= -1 + ind AND b(j) = x))
-->                 AND ((exists j: 1 <= j AND j <= len AND b(j) = x))
-->                 AND (x = x_i) AND (len = len_i)
-->            ==> (1 - ind + len >= 1 - ind + len)
--> Result    : proved
WHILE b(ind) /= x LOOP
   ind := ind + 1;
END LOOP;
```

17. nested_for

Computing the product of the natural numbers a and b by repeated addition using two nested FOR-loops.

Input to NPPV

```
{ a >= 0 and b >= 0 }
BEGIN
p := 0;
{ a >= 0 and p = 0 and b >= 0 }
for i := 1 to b do { p = (i-1)*a and a >= 0 }        +++   FOR
  for j := 1 to a do { p = (i-1)*a+j-1 and a >= 0 }
    p := p+1
END
{ p = b*a }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

a>=0 AND b>=0
     ==>
          a>=0 AND 0=0 AND b>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

a>=0 AND p=0 AND b>=0
     ==>
          p=(1-1)*a AND a>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

p=(b+1-1)*a AND a>=0
     ==>
          p=b*a
=========  Proof succeeded  =========
=====================================
=====================================
=== Verification Condition No.: 4 ===

a>=0 AND p=0 AND b>=0 AND b<1
     ==>
          p=b*a
--------- Remains to prove  ---------
0<=a AND 0<=b AND b<1
     ==>
          0=a*b
=====================================

=====================================
=== Verification Condition No.: 5 ===

p=(i-1)*a AND a>=0 AND 1<=i AND i<=b
     ==>
          p=(i-1)*a+1-1 AND a>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 6 ===

p=(i-1)*a+a+1-1 AND a>=0
     ==>
          p=(i+1-1)*a AND a>=0
--------- Remains to prove  ---------
0<=a
     ==>
          (i-1)*a+a=a*i
=====================================

=====================================
=== Verification Condition No.: 7 ===

p=(i-1)*a AND a>=0 AND 1<=i AND i<=b AND a<1
     ==>
          p=(i+1-1)*a AND a>=0
--------- Remains to prove  ---------
0<=a AND 1<=i AND i<=b AND a<1
     ==>
          (i-1)*a=a*i
=====================================

=====================================
=== Verification Condition No.: 8 ===

p=(i-1)*a+j-1 AND a>=0 AND 1<=j AND j<=a
     ==>
          p+1=(i-1)*a+j+1-1 AND a>=0
=========  Proof succeeded  =========
=====================================
```

Input to FPP

```
   --  Example 17
   --!pre: a>=0 and b>=0 and a = a_i and b = b_i;
p := 0;
   --!pre : p=0 and a>=0 and b>=0 and a = a_i and b = b_i;
   --!post: p=b*a and a = a_i and b = b_i;
   --!inv : p=i*a and a>=0 and a = a_i and b = b_i;
FOR i IN 1..b LOOP
      --!pre : p = (i-1)*a and a>=0;
      --!post: p = i*a;
      --!inv : p = (i-1)*a+j;
   FOR j IN 1..a LOOP
      p := p+1;
   END LOOP;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 9:14
The answer to your query is:

--!pre       : (a >= 0 AND b >= 0 AND a = a_i AND b = b_i)
--> wp       : (a >= 0 AND b >= 0 AND a = a_i AND b = b_i)
--> vc       : (True)
--> Result: proved
p := 0;

--!pre       : (p = 0 AND a >= 0 AND b >= 0 AND a = a_i AND b = b_i)
--!post      : (p = a*b AND a = a_i AND b = b_i)
--!inv       : (p = a*i AND a >= 0 AND a = a_i AND b = b_i)
-->functionality -------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :     (1 <= b AND p = 0 AND a >= 0 AND b >= 0 AND a = a_i AND b = b_i)
-->              ==> (p = 0 AND a >= 0 AND a = a_i AND b = b_i)
--> Result    : proved
-->induction :     (1 <= b AND p = a*(-1 + i) AND a >= 0 AND a = a_i AND b = b_i)
-->              ==>          (1 <= a) AND (p = a*(-1 + i)) AND (a >= 0)
-->                      AND (forall(($0)),
-->                               (p = a*(-1 + i) AND a >= 0 ==> $0 = a*i)
-->                           ==> ($0 = a*i AND a >= 0 AND a = a_i AND b = b_i))
-->                  OR (1 >= 1 + a AND p = a*i AND a >= 0 AND a = a_i AND b = b_i)
--> Result    : proved
-->final     :     (1 <= b AND p = a*b AND a >= 0 AND a = a_i AND b = b_i)
-->              ==> (p = a*b AND a = a_i AND b = b_i)
--> Result    : proved
-->null loop :          (1 >= 1 + b) AND (p = 0) AND (a >= 0) AND (b >= 0)
-->                  AND (a = a_i) AND (b = b_i)
-->              ==> (p = a*b AND a = a_i AND b = b_i)
--> Result    : proved
FOR i IN 1 .. b  LOOP

   --!pre       : (p = a*(-1 + i) AND a >= 0)
   --!post      : (p = a*i)
   --!inv       : (p = j + a*(-1 + i))
   -->functionality -------------------------
   -->func      : (initial AND induction AND final AND null loop)
   -->initial   :(1 <= a AND p = a*(-1 + i) AND a >= 0 ==> p = a*(-1 + i))
   --> Result    : proved
   -->induction :(1 <= a AND p = -1 + j + a*(-1 + i) ==> 1 + p = j + a*(-1 + i))
   --> Result    : proved
   -->final     :(1 <= a AND p = a + a*(-1 + i) ==> p = a*i)
   --> Result    : proved
   -->null loop :(1 >= 1 + a AND p = a*(-1 + i) AND a >= 0 ==> p = a*i)
   --> Result    : proved
   FOR j IN 1 .. a  LOOP
      p := p + 1;
   END LOOP;

END LOOP;
```

18. proof

An example that shows that program verification is as difficult as theorem proving [Gum 99a]. P(n) is a property of the natural numbers. P(n) is true, if and only if the following program ”proof“ never terminates. In this example the program provers are used for formula manipulation. Since P(k) and I(k) are only abstract functions no proof is possible but the program prover generates the conditions which P(k) and I(k) must fulfill.

Input to NPPV

```
{ true }
BEGIN
  k := 0;
  WHILE P(k) DO { I(k) }
```

```
    k := k+1
END
{ false }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

TRUE
     ==>
           I(0)
--------- Remains to prove  ---------
I(0)
=====================================

=====================================
=== Verification Condition No.: 2 ===

I(k)
     ==>
           I(k)
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

I(k) AND NOT P(k)
     ==>
           FALSE
--------- Remains to prove  ---------
I(k)
     ==>
           P(k)
=====================================

=====================================
=== Verification Condition No.: 4 ===

I(k) AND P(k)
     ==>
           I(k+1)
--------- Remains to prove  ---------
I(k) AND P(k)
     ==>
           I(k+1)
=====================================
```

FPP: in this abstract example it is not possible to give a termination function for the WHILE-loop. Since FPP requires such a termination function the example is not appropriate for FPP.

19. quad

Computing the square n of a positive integer k as the sum of the first k odd numbers .

Input to NPPV

```
{ k >= 1}
BEGIN
n := 0; { k >= 1 AND n = 0 }
FOR i := 0 TO k-1 DO { n = i*i }
  BEGIN
    j := 2*i+1;
    n := n+j
  END
END
{ n = k*k }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

k>=1
     ==>
           k>=1 AND 0=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

k>=1 AND n=0
     ==>
           n=0*0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

n=(k-1+1)*(k-1+1)
     ==>
           n=k*k
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

k>=1 AND n=0 AND k-1<0
     ==>
           n=k*k
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 5 ===

n=i*i AND 0<=i AND i<=k-1
     ==>
           n+2*i+1=(i+1)*(i+1)
--------- Remains to prove  ---------
0<=i AND i+1<=k
     ==>
           2*i+1=i+i+1
=====================================
```

Input to FPP

```
   --  Example 19
   --!pre: n >= 0 and n = n_i;
s := 0;
   --!pre : n >= 0 and s = 0 and n = n_i;
   --!post: s = n**2 and n = n_i;
   --!inv : s = (i+1)**2 and n = n_i;
FOR i IN 0 .. n-1 LOOP
   j := 2*i+1;
   s := s+j;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 10:01
The answer to your query is:

--!pre       : (n >= 0 AND n = n_i)
--> wp       : (n >= 0 AND n = n_i)
--> vc       : (True)
--> Result: proved
s := 0;

--!pre       : (n >= 0 AND s = 0 AND n = n_i)
--!post      : (s = n**2 AND n = n_i)
--!inv       : (s = (1 + i)**2 AND n = n_i)
-->functionality -------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :(0 <= -1 + n AND n >= 0 AND s = 0 AND n = n_i ==> s = 0 AND n = n_i)
--> Result    : proved
-->induction :     (0 <= -1 + n AND 1 = i**2 AND n = n_i)
-->               ==> (3 + 2*i = (1 + i)**2 AND n = n_i)
--> Result    : proved
-->final     :(0 <= -1 + n AND s = n**2 AND n = n_i ==> s = n**2 AND n = n_i)
--> Result    : proved
-->null loop :(0 >= n AND n >= 0 AND s = 0 AND n = n_i ==> s = n**2 AND n = n_i)
--> Result    : proved
FOR i IN 0 .. n - 1  LOOP
   j := 2 * i + 1;
   s := s + j;
END LOOP;
```

20. root   Computing the floor of the square root of a nonnegative integer.

Input to NPPV

```
{ a = 0 and n >= 0 }
WHILE (a+1)*(a+1) <= n DO [n-a] { a*a <= n }
  a := a+1
{ (a+1)*(a+1) > n and n >= a*a }
```

Output of NPPV

```
=====================================
=== Verification Condition No.: 1 ===

a=0 AND n>=0
     ==>
           a*a<=n
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 2 ===

(a+1)*(a+1)<=n AND a=0 AND n>=0
     ==>
           n-a>=0
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 3 ===

a*a<=n AND NOT (a+1)*(a+1)<=n
     ==>
           (a+1)*(a+1)>n AND n>=a*a
=========  Proof succeeded  =========
=====================================

=====================================
=== Verification Condition No.: 4 ===

(a+1)*(a+1)<=n AND a*a<=n AND n-a>=0
     ==>
           (a+1)*(a+1)<=n AND n-(a+1)>=0
--------- Remains to prove  ---------
a*a+a+a+1<=n AND a*a<=n AND a<=n
```

```
     ==>
           a+1<=n
=====================================

=====================================
=== Verification Condition No.: 5 ===

(a+1)*(a+1)<=n AND a*a<=n AND n-a=n-a
     ==>
           n-(a+1)<n-a
=========  Proof succeeded  =========
=====================================
```

Input to FPP

```
   --  Example 20
   --!Pre :  a=0 and n>=0 and n = n_i;
   --!Post: (a+1)**2>n and n>=a**2 and n = n_i;
   --!Inv : a**2<=n and n = n_i;
   --!term: n-a;
WHILE (a+1)**2 <= n LOOP a := a+1; END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 10:13
The answer to your query is:

--!pre       : (a = 0 AND n >= 0 AND n = n_i)
--!post      : ((1 + a)**2 >= 1 + n AND n >= a**2 AND n = n_i)
--!inv       : (a**2 <= n AND n = n_i)
--!term      : (-a + n)
-->functionality --------------------------
-->initial   :(a = 0 AND n >= 0 AND n = n_i ==> a**2 <= n AND n = n_i)
--> Result    : proved
-->induction :     ((1 + a)**2 <= n AND a**2 <= n AND n = n_i)
-->             ==> ((1 + a)**2 <= n AND n = n_i)
--> Result    : proved
-->final     :     ((1 + a)**2 >= 1 + n AND a**2 <= n AND n = n_i)
-->             ==> ((1 + a)**2 >= 1 + n AND n >= a**2 AND n = n_i)
--> Result    : proved
-->termination --------------------------
-->initial   :((1 + a)**2 <= n AND a**2 <= n AND n = n_i ==> -a + n >= 1)
--> Result    : proved
-->induction :((1 + a)**2 <= n AND a**2 <= n AND n = n_i ==> -a + n >= -a + n)
--> Result    : proved
WHILE (a + 1) ** 2 <= n LOOP
   a := a + 1;
END LOOP;
```

21. swap1   Swapping the values of two variables using an auxiliary variable.

Input to NPPV

```
{ x = A and y = B }
BEGIN
  temp := x ;
  x    := y ;
  y    := temp
END
{ x = B and y = A }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

x=A AND y=B
     ==>
           y=B AND x=A
=========  Proof succeeded  =========
=====================================
```

Input to FPP

```
   --  Example 21
   --!pre: x = x_i and y = y_i;
temp := x ;
x    := y ;
y    := temp;
   --!post: x = y_i and y = x_i;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 10:24
The answer to your query is:

--!pre       : (x = x_i AND y = y_i)
--> wp       : (y = y_i AND x = x_i)
--> vc       : (x = x_i AND y = y_i ==> y = y_i AND x = x_i)
--> Result: proved
temp := x;
x := y;
y := temp;
--!post      : (x = y_i AND y = x_i)
```

22. swap2

Tricky but unsafe version of swapping the values of two variables without an auxiliary variable.

Input to NPPV

```
{ x = M and y = N }
BEGIN
  x := x - y;
  y := x + y;
  x := y - x
END
{ x = N and y = M }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

x=M AND y=N
     ==>
           x-y+y-(x-y)=N AND x-y+y=M
=========  Proof succeeded  =========
=====================================
```

Input to FPP

```
   --  Example 22
   --!pre : x = x_i and y = y_i;
x := x - y;
y := x + y;
x := y - x;
   --!post: x = y_i and y = x_i;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 10:31
The answer to your query is:

--!pre       : (x = x_i AND y = y_i)
--> wp       : (y = y_i AND x = x_i)
--> vc       : (x = x_i AND y = y_i ==> y = y_i AND x = x_i)
--> Result: proved
x := x - y;
y := x + y;
x := y - x;
--!post      : (x = y_i AND y = x_i)
```

Since neither NPPV nor FPP take the limited ranges of integer types into account both say that the programs are correct. In typical implementations of Pascal and Ada the programs are not correct, because the difference in ”x := x-y;“ cannot be computed for all legal combinations of x and y [Win 90].

23. swap2ty   The same as example 22 but with type checking assertions.

Input to NPPV

```
   { x = M and y = N and -100 <= x and x <= 100 and -100 <= y and y <= 100 }
   BEGIN
     x := x - y;
   {  -100 <= x and x <= 100 and -100 <= y and y <= 100 }
     y := x + y;
   {  -100 <= x and x <= 100 and -100 <= y and y <= 100 }
     x := y - x

   END
   { x = N and y = M and -100 <= x and x <= 100 and -100 <= y and y <= 100 }
```

Output from NPPV

```
     =====================================
     === Verification Condition No.: 1 ===

     x=M AND y=N AND -100<=x AND x<=100 AND -100<=y AND y<
     =100
              ==>
                     -100<=x-y AND x-y<=100 AND -100<=y AN
     D y<=100

     --------- Remains to prove  ---------
     -100<=M AND M<=100 AND -100<=N AND N<=100
              ==>
                     -100+N<=M AND M<=100+N
     =====================================

     =====================================
     === Verification Condition No.: 2 ===

     -100<=x AND x<=100 AND -100<=y AND y<=100
              ==>
                     -100<=x AND x<=100 AND -100<=x+y AND
     x+y<=100

     --------- Remains to prove  ---------
     -100<=x AND x<=100 AND -100<=y AND y<=100
              ==>
```

```
                -100<=x+y AND x+y<=100
=====================================

=====================================
=== Verification Condition No.: 3 ===

-100<=x AND x<=100 AND -100<=y AND y<=100
         ==>
                y-x=N AND y=M AND -100<=y-x AND y-x<=
100 AND -100<=y AND y<=100

--------- Remains to prove  ---------
-100<=x AND x<=100 AND -100<=y AND y<=100
         ==>
                y=M AND y-x=N AND -100+x<=y AND y<=10
0+x
=====================================
```

Input to FPP

```
      --  Example 23
      --!pre : x=x_i and -100 <= x and x <= +100 and
      --!pre : y=y_i and -100 <= y and y <= +100;
   x := x - y;
      --!post: -100 <= x and x <= +100 and -100 <= y and y <= +100;
   y := x + y;
      --!post: -100 <= x and x <= +100 and -100 <= y and y <= +100;
   x := y - x;
      --!post: x=y_i and -100 <= x and x <= +100 and
      --!post: y=x_i and -100 <= y and y <= +100;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.24, 12:28
The answer to your query is:

--!pre       :     (x = x_i) AND (-100 <= x) AND (x <= 100)
-->              AND (y = y_i) AND (-100 <= y) AND (y <= 100)
--> wp       : (-100 <= x - y AND x - y <= 100 AND -100 <= y AND y <= 100)
--> vc       :          (x = x_i) AND (-100 <= x) AND (x <= 100)
-->                   AND y <= 100) AND (y = y_i) AND (-100 <= y) AND (y <= 100)
-->              ==> (-100 <= x - y AND x - y <= 100 AND -100 <= y AND y <= 100)
--> Result: not proved
--> fc       :     (-100 + x_i - y >= 1) AND (100 - x_i >= 0) AND (100 + x_i >= 0)
-->              AND (100 + y >= 0) AND (100 - y >= 0)

x := x - y;
--!post      : (-100 <= x AND x <= 100 AND -100 <= y AND y <= 100)
--> wp       : (-100 <= x AND x <= 100 AND -100 <= x + y AND x + y <= 100)
--> vc       :     (-100 <= x AND x <= 100 AND -100 <= y AND y <= 100)
-->              ==> (-100 <= x AND x <= 100 AND -100 <= x + y AND x + y <= 100)
--> Result: not proved
--> fc       :     (-100 - x - y >= 1) AND (100 + x >= 0) AND (100 - y >= 0)
-->              AND (100 + y >= 0) AND (100 - x >= 0)

y := x + y;
--!post      : (-100 <= x AND x <= 100 AND -100 <= y AND y <= 100)
--> wp       :     (-x + y = y_i) AND (-100 <= -x + y) AND (-x + y <= 100)
-->              AND (y = x_i) AND (-100 <= y) AND (y <= 100)
--> vc       :      (-100 <= x AND x <= 100 AND -100 <= y AND y <= 100)
-->              ==>      (-x + y = y_i) AND (-100 <= -x + y) AND (-x + y <= 100)
-->                   AND (y = x_i) AND (-100 <= y) AND (y <= 100)
--> Result: not proved
--> fc       :      (100 - x >= 0) AND (100 + x >= 0) AND (100 + y >= 0)
-->               AND (y /= x_i) AND (100 - y >= 0)

x := y - x;
--!post      :     (x = y_i) AND (-100 <= x) AND (x <= 100)
-->              AND (y = x_i) AND (-100 <= y) AND (y <= 100)
```

When the limited domains of integer types are checked both provers say that the programs are not correct. The first falsification condition in the output from FPP is equivalent to

$$-100 \le x \le +100 \ \wedge -100 \le y \le +100 \ \wedge \ x - y \ge 101$$

which is satisfied by e.g. $x = 100 \ \wedge . \ y = -1$ which is a legal pair of values for x and y.

24. swap2ty2 The same as example 23 but with sufficiently strong preconditions.

Input to NPPV

```
{ x = M and y = N and -100 <= x-y and x-y <= 100 and -100 <= x and x <= 100 and -100
<= y and y <= 100 }
BEGIN
  x := x - y;
{ x = M-N and y = N and -100 <= x+y and x+y <= 100 and -100 <= x and x <= 100 and -100
<= y and y <= 100 }
  y := x + y;
{ x = M-N and y = M and -100 <= y-x and y-x <= 100 and -100 <= x and x <= 100 and -100
<= y and y <= 100 }
  x := y - x
END
{ x = N and y = M and -100 <= x and x <= 100 and -100 <= y and y <= 100 }
```

Output from NPPV

```
=== Verification Condition No.: 1 ===

x=M AND y=N AND -100<=x-y AND x-y<=100 AND -100<=x AN
D x<=100 AND -100<=y AND y<=100
         ==>
                x-y=M-N AND y=N AND -100<=x-y+y AND x
-y+y<=100 AND -100<=x-y AND x-y<=100 AND -100<=y AND
y<=100

=========  Proof succeeded  =========

=====================================
=== Verification Condition No.: 2 ===

x=M-N AND y=N AND -100<=x+y AND x+y<=100 AND -100<=x
AND x<=100 AND -100<=y AND y<=100
         ==>
                x=M-N AND x+y=M AND -100<=x+y-x AND x
+y-x<=100 AND -100<=x AND x<=100 AND -100<=x+y AND x+
y<=100

--------- Remains to prove  ---------
-100<=M AND M<=100 AND -100+N<=M AND M<=100+N AND -10
0<=N AND N<=100
         ==>
                -100<=M+N+N-M-N AND M+N+N-M-N<=100
=====================================

=====================================
=== Verification Condition No.: 3 ===

x=M-N AND y=M AND -100<=y-x AND y-x<=100 AND -100<=x
AND x<=100 AND -100<=y AND y<=100
         ==>
                y-x=N AND y=M AND -100<=y-x AND y-x<=
100 AND -100<=y AND y<=100

--------- Remains to prove  ---------
-100+M-N<=M AND M<=100+M-N AND -100+N<=M AND M<=100+N
 AND -100<=M AND M<=100
         ==>
                M+N-M=N
=====================================
```

Input to FPP

```
   -- Example 24a
   --!pre : x = x_i and y = y_i and -100 <= x-y and x-y <= +100 and
   --!pre : -100 <= x and x <= +100 and -100 <= y and y <= +100;
x := x - y;
   --!pre : x = x_i-y_i and y = y_i and -100 <= x+y and x+y <= +100 and
   --!pre : -100 <= x and x <= +100 and -100 <= y and y <= +100;
y := x + y;
   --!pre : x = x_i-y_i and y = x_i and -100 <= y-x and y-x <= +100 and
   --!pre : -100 <= x and x <= +100 and -100 <= y and y <= +100;
x := y - x;
   --!post: x = y_i and -100 <= x and x <= +100 and
   --!post: y = x_i and -100 <= y and y <= +100;
```

Output fromFPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.25, 9:04
The answer to your query is:

--!pre     :     (x = x_i) AND (y = y_i) AND (-100 <= x - y) AND (x - y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= y) AND (y <= 100)
--> wp     :     (x - y = x_i - y_i) AND (y = y_i) AND (-100 <= x) AND (x <= 100)
-->            AND (-100 <= x - y) AND (x - y <= 100) AND (-100 <= y) AND (y <= 100)
--> vc     :          (x = x_i) AND (y = y_i) AND (-100 <= x-y) AND (x-y <= 100)
-->                 AND (-100 <= x) AND (x <= 100) AND (-100 <= y) AND (y <= 100)
-->            ==>      (x-y = x_i-y_i) AND (y = y_i) AND (-100 <= x) AND (x <= 100)
-->                 AND (-100 <= x-y) AND (x-y <= 100) AND (-100 <= y) AND (y <= 100)
--> Result: proved
x := x - y;
--!pre     :     (x = x_i - y_i) AND (y = y_i) AND (-100 <= x+y) AND (x+y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= y) AND (y <= 100)
--> wp     :     (x = x_i - y_i) AND (x + y = x_i) AND (-100 <= y) AND (y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= x + y) AND (x + y <= 100)
--> vc     :     (x = x_i - y_i) AND (y = y_i) AND (-100 <= x+y) AND (x+y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= y) AND (y <= 100)
-->     ==>      (x = x_i - y_i) AND (x + y = x_i) AND (-100 <= y) AND (y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= x + y) AND (x + y <= 100)
--> Result: proved
y := x + y;
--!pre     :     (x = x_i - y_i) AND (y = x_i) AND (-100 <= -x+y) AND (-x+y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= y) AND (y <= 100)
--> wp     :     (-x + y = y_i) AND (-100 <= -x+y) AND (-x+y <= 100)
-->            AND (y = x_i) AND (-100 <= y) AND (y <= 100)
--> vc     :     (x = x_i - y_i) AND (y = x_i) AND (-100 <= -x+y) AND (-x+y <= 100)
-->            AND (-100 <= x) AND (x <= 100) AND (-100 <= y) AND (y <= 100)
-->      ==>     (-x + y = y_i) AND (-100 <= -x + y) AND (-x + y <= 100)
-->            AND (y = x_i) AND (-100 <= y) AND (y <= 100)
--> Result: proved
x := y - x;
--!post    :     (x = y_i) AND (-100 <= x) AND (x <= 100)
-->            AND (y = x_i) AND (-100 <= y) AND (y <= 100)
```

25. swap3

This example is a generalization of the examples 22 - 24. As in example 18 the program provers are used for formula manipulation to generate the conditions, which the three function s, g and h must fulfill, for the program to be correct [Gum 99a]. It is not checked, whether the results of these functions lie in the types of the variables.

Input to NPPV

```
{ x = M and y = N }
BEGIN
  y := s(x,y);
```

```
    x := g(x,y);
    y := h(y,x)
  END
  { x = N and y = M }
```

Output from NPPV

```
  =====================================
  === Verification Condition No.: 1 ===

  x=M AND y=N
       ==>
            g(x,s(x,y))=N AND h(s(x,y),g(x,s(x,y)))=M
  --------- Remains to prove  ---------
  g(M,s(M,N))=N
  --- and also ---
  h(s(M,N),g(M,s(M,N)))=M
  =====================================
```

Input to FPP

```
   --  Example 25
   --!pre: x = x_i and y = y_i;
x := f(x,y);
y := g(x,y);
x := h(x,y);
   --!post: x = y_i and y = x_i;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.09.25, 9:59
The answer to your query is:

--!pre       : (x = x_i AND y = y_i)
--> wp       : (h(f(x,y),g(f(x,y),y)) = y_i AND g(f(x,y),y) = x_i)
--> vc       :      (x = x_i AND y = y_i)
-->            ==> (h(f(x,y),g(f(x,y),y)) = y_i AND g(f(x,y),y) = x_i)
--> Result: not proved
--> fc       : (Not(g(f(x_i,y),y) = x_i AND h(f(x_i,y),g(f(x_i,y),y)) = y))

x := f(x, y);
y := g(x, y);
x := h(x, y);
--!post      : (x = y_i AND y = x_i)
```

26. Cube

Compute the cube of the nonnegative integer n using additions only [GH 99].

Input to NPPV

```
{ N >= 0 and N = N_i}

BEGIN
  x := 0;
  y := 1;
  z := 6;
  { N >= 0 and N = N_i and x=0 and y=1 and z=6 }
  FOR i := 1 to N DO  { x = (i-1)*(i-1)*(i-1) and y = 3*(i-1)*(i-1)+3*(i-1)+1
                        and z = 6*(i-1)+6 and N = N_i }
    BEGIN
      x := x+y;
      y := y+z;
      z := z+6
    END
END
{ x = N*N*N and N = N_i }
```

Output from NPPV

```
=====================================
=== Verification Condition No.: 1 ===

N>=0 AND N=N_i
         ==>
                N>=0 AND N=N_i AND 0=0 AND 1=1 AND 6=6

=========  Proof succeeded  =========
=====================================


=====================================
=== Verification Condition No.: 2 ===

N>=0 AND N=N_i AND x=0 AND y=1 AND z=6
         ==>
                x=(1-1)*(1-1)*(1-1) AND y=3*(1-1)*(1-
1)+3*(1-1)+1 AND z=6*(1-1)+6 AND N=N_i

=========  Proof succeeded  =========
=====================================

=====================================

=== Verification Condition No.: 3 ===

x=(N+1-1)*(N+1-1)*(N+1-1) AND y=3*(N+1-1)*(N+1-1)+3*(
N+1-1)+1 AND z=6*(N+1-1)+6 AND N=N_i
         ==>
                x=N*N*N AND N=N_i

=========  Proof succeeded  =========
=====================================


=====================================
=== Verification Condition No.: 4 ===

N>=0 AND N=N_i AND x=0 AND y=1 AND z=6 AND N<1
         ==>
                x=N*N*N AND N=N_i

--------- Remains to prove  ---------
0<=N AND N=N_i AND N<1
         ==>
                0=N*N*N
=====================================


=====================================
=== Verification Condition No.: 5 ===

x=(i-1)*(i-1)*(i-1) AND y=3*(i-1)*(i-1)+3*(i-1)+1 AND
 z=6*(i-1)+6 AND N=N_i AND 1<=i AND i<=N
         ==>
                x+y=(i+1-1)*(i+1-1)*(i+1-1) AND y+z=3
*(i+1-1)*(i+1-1)+3*(i+1-1)+1 AND z+6=6*(i+1-1)+6 AND
N=N_i

--------- Remains to prove  ---------
N=N_i AND 1<=i AND i<=N
         ==>
                (i-1)*i*i+(1-i)*i+(3*i-3)*i+3+3*i+1-3
*i-3-((i-1)*i+1-i)=i*i*i AND (3*i-3)*i+3+3*i+1+6*i-3-
3*i=3*i*i+3*i+1
=====================================
```

Input to FPP

```
   --  Example 26
   --!pre: n >= 0 and n = n_i;
x := 0;
y := 1;
z := 6;
   --!pre :     n >= 0 and n = n_i
   --!pre : and x=0 and y=1 and z=6;
   --!post: x=n**3 and n = n_i;
   --!inv :     x=i**3 and y=3*i**2+3*i+1
   --!inv : and z=6*i+6 and n = n_i;
FOR i IN 1 .. n LOOP
   x := x+y;
   y := y+z;
   z := z+6;
END LOOP;
```

Output from FPP

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1999.10.05, 8:46
The answer to your query is:

--!pre       : (n >= 0 AND n = n_i)
--> wp       : (n >= 0 AND n = n_i)
--> vc       : (True)
--> Result: proved
x := 0;
y := 1;
z := 6;

--!pre       : (n >= 0 AND n = n_i AND x = 0 AND y = 1 AND z = 6)
--!post      : (x = n**3 AND n = n_i)
--!inv       : (x = i**3 AND y = 1 + 3*i + 3*i**2 AND z = 6 + 6*i AND n = n_i)
-->functionality ----------------------------
-->func      : (initial AND induction AND final AND null loop)
-->initial   :     (1 <= n AND n >= 0 AND n = n_i AND x = 0 AND y = 1 AND z = 6)
-->            ==> (x = 0 AND y = 1 AND z = 6 AND n = n_i)
--> Result    : proved
-->induction :          (1 <= n) AND (x = (-1 + i)**3)
-->                 AND (y = 1 + 3*(-1 + i) + 3*(-1 + i)**2)
-->                 AND (z = 6 + 6*(-1 + i)) AND (n = n_i)
-->            ==>      (x + y = i**3) AND (y + z = 1 + 3*i + 3*i**2)
-->                 AND (6 + z = 6 + 6*i) AND (n = n_i)
--> Result    : proved
-->final     :          (1 <= n) AND (x = n**3)
-->                 AND (y = 1 + 3*n + 3*n**2)
-->                 AND (z = 6 + 6*n) AND (n = n_i)
-->            ==> (x = n**3 AND n = n_i)
--> Result    : proved
-->null loop :     (1 >= 1 + n AND n >= 0 AND n = n_i AND x = 0 AND y = 1 AND z =
6) -->            ==> (x = n**3 AND n = n_i)
--> Result    : proved
FOR i IN 1 .. n  LOOP
   x := x + y;
   y := y + z;
   z := z + 6;
END LOOP;
```