**FRIEDRICH-SCHILLER-
UNIVERSITÄT JENA**

FORSCHUNGSERGEBNISSE

Der Fakultät für Mathematik und Informatik

Eingang: 1998.11.07 **Nr. Math / Inf / 98 / 13** **Als Manuskript gedruckt**

NEW PROOF RULES FOR FOR-LOOPS

Jürgen F. H. Winkler

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

NEW PROOF RULES FOR FOR-LOOPS

Jürgen F. H. Winkler

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

The FOR-loop is an element of many programming languages and is used in many programs. Despite this fact, it is treated quite rarely in the literature on program verification. In this paper we develop a new proof rule for FOR-loops. The derivation of the proof rule is done in two steps: (1) transform the FOR-loop into an equivalent ersatz program, and (2) derive the proof rules or verification conditions from the ersatz program.
The main difference between the new rules and earlier proof rules for FOR-loops is that the new proof rules do not require the invariant to hold before the first execution of the loop body and that the empty loop (zero repetitions) is also covered. This simplifies the proof of certain loops. A second difference is that the proof rules work for FOR-loops as they occur in existing programming languages, especially languages like Ada or Pascal. This means that the rule can be readily applied by the program developer. In the literature on program verification often some idealized form of FOR-loop is used, e.g. one in which the variables occurring in the iteration clauses cannot be modified within the loop body. The only restriction of the new proof rule is that the evaluation of the expressions in the iteration clause may not have any side effects.

„A *cycle* of operations, then, must be understood to signify any *set of operations* which is repeated *more than once*." Ada Augusta Lovelace, 1843

„Ausser dieser allgemeinen W-Vorschrift werden noch einige spezielle, die besonders häufig vorkommen, eingeführt."[1] Konrad Zuse, 1945

„Situations in which arrays are scanned strictly sequentially are so frequent that a special notation is introduced to denote such actions." Alagic and Arbib, 1978

„We are always allowed to invent new syntax if we explain the rules for its use." Eric C. R. Hehner, 1993

1 Introduction

Repetition is an essential element of computation. Repetition can be done by loops, by goto, and by recursion. Different forms of loops have been developed in programming languages. The main forms are the prechecked loop (WHILE), the postchecked loop (REPEAT), and the controlled loop (FOR). Of these the WHILE-loop is the most general, but as the quotation of Alagic and Arbib above indicates, there are many situations in which the FOR-loop is adequate. This has already been observed by Zuse in 1945, when he was developing the Plankalkül. He first introduces the WHILE-loop [Zus 72: 4.32] and then mentions that for special situations more specific loop constructs could be used. Among these he mentions also the FOR-loop [Zus 72: 4.33] (he does not use the keyword FOR). A FOR-loop like construct is also mentioned by Ada A. Lovelace in her notes on Menabreas paper on the Analytical Engine of Charles Babbage [Men 55: 392, 394].

The FOR-loop is an element of many programming languages. The Plankalkül has already been mentioned. Of the other early high level languages, which were developed in the late fifties, Fortran [Bac 81], Algol 60 [BBG 63], and COBOL [Sam 69: 356] contain the FOR-loop. LISP [McC 81] uses recursion and does therefore not have loop constructs. We do not mention all those more recent programming languages which contain the FOR-loop.

[1] „Besides this general W-instruction, some other special ones are introduced for frequently recurring cases." [Zus 89: 63]

In this paper we look at the verification of FOR-loops and propose a new verification scheme for FOR-loops. This scheme consists of two steps: (1) replace the FOR-loop by an ersatz program, and (2) derive proof rules or verification conditions from the ersatz program. Based on this scheme new proof rules for FOR-loops can be derived which may be easier to apply and which cover also rather exotic cases. Such rules are needed because the FOR-loop is used quite often in practical programming, but is treated quite rarely in the work on program verification.

Proof rules or verification conditions allow us to show that a program (fragment) P fulfills a specification S. A specification may be given as a pair of assertions: S = (pre, post). Together, P and S form a spec-prog (= specified program) SP = {pre} P {post}. The fact that P fulfills S or that P and S are consistent can be expressed as [pre $\Rightarrow$ wp(P, post)], where wp is the weakest precondition and [...] means „in all program states“ [DS 90: 8]. More abstractly, consistency between S and P means that P is a refinement of S [Win 96: 15]. In the rest of the paper we mostly use the term „consistency between S and P“ or simply „consistency of a spec-prog SP“. This terminology takes into account that in case of inconsistency, i.e. when consistency cannot be proved, we can neither blame P nor S as the sole reason for the inconsistency. We only know that the spec-prog is not consistent. The term „proof of a program P wrt a specification S“ on the other hand, is more asymmetric: it suggests that the specification S is assumed to be correct and that in case of inconsistency the program P is to blame. In practical development we have also the situation that the specification does not mention all necessary facts to show the consistency of SP, but that the program P does the job. As a consequence we use „to show the consistency of a spec-prog (S, P)“ instead of „verification of P wrt to S“, and „consistency condition (CC)“ instead of „proof rule“. By „verification“ we now mean „to show the consistency“. We still use sometimes the traditional terminology, especially when referring to the existing literature.

When wp(P, post) can be computed the verification can be done exactly. Sometimes (e.g. in the case of certain loops) wp(P, post) cannot be computed efficiently but some other scheme can be used to show some weaker form of consistency. One example for this is the scheme for proving specified WHILE-loops using an invariant and a termination function [Gri 83: 145]. More abstractly, this means we are using some condition pc(P, post) which implies wp(P, post). If [pre $\Rightarrow$ pc(P, post)] then we have shown the consistency of {pre} P {post}.

In this paper we use FOR-loops of existing languages and not some idealized form. The main reason for this is that our goal is to provide consistency conditions, which can be used by the practicing software developer. Up to now, this is usually not done in the literature on program verification. On the other hand, engineers in other branches of engineering typically find in their literature formulas which are ready to be used [BK 94; Dor 93]. [PR 97; Tuc 97] do not contain proof rules for FOR-loops. The FOR-loops used in this paper are those from Pascal and Ada. Ada especially seems to be a good candidate for program verification because it is mainly used in critical applications.

The paper is organized as follows. In section 2 we review existing proof rules for specified FOR-loops. Section 3 introduces the ersatz program for the FOR-loop which is used to define the semantics of the FOR-loop. Based on this ersatz program we derive new proof rules for specified FOR-loops in section 4. Section 5 contains specific rules for Ada FOR-loops and section 6 contains some examples for the use of the new rules. In section 7 we transform a FOR-loop into an equivalent WHILE-loop and derive a consistency condition for the specified FOR-loop. This way is somewhat more complicated than the derivation via the ersatz program introduced in section 3.

2 Existing Proof Rules for the FOR-Loop

When the FOR-loop is used in serious program construction there should be rules at hand to prove the consistency of a specification and such a loop. The literature on program verification typically discusses and gives proof rules for the WHILE-loop. On the other hand, the FOR-loop is treated very rarely. [Abr 96; AO 94; Bab 87; Bab 91; Bes 95; Cou 92; Dij 76; Fra 92; Fut 89; Gri 83; Hoa 69] do not discuss the FOR-loop. It is discussed in [AA 78; Dah 92; Flo 67; Heh 93; Hoa 72; HW 73]. [Flo 67] is the first to give a verification condition for the FOR-loop. He discusses the very general FOR-loop of Algol60 which does not adhere to the limitations of modern FOR-loops as they are mentioned in sec. 3. [Hoa 72] gives proof rules for FOR-

loops similar to the loops we discuss in this paper. These rules are also used in [AA 78; HW 93]. [AA 78: 81..84] contains also a reduction of the FOR-loop to a combination of IF, SEQ and WHILE, but they do not mention that the loop parameter and the bounds must not be changed/written inside the loop body. [Dah 92] uses a more complicated rule which is derived from a more complicated scheme for the definition of the semantics of the FOR-loop. [Heh 93] uses a different approach, in that he expresses statements as predicates over the pre- and post-states. For the FOR-loop he does not give a direct definition but a refinement rule. This approach is rather different from the approach used in this paper.

[Hoa 72] gives the following proof rule for the FOR-loop:

$$\frac{a \le x \le b \;\;\&\;\; I([a \ldots x)) \; \{Q\} \; I([a \ldots x])}{I([]) \;\; \{\textbf{for}\;\; x := a \;\; \textbf{to}\; b \; \textbf{do}\; Q\} \;\; I([a \ldots b])}$$

with the following restrictions:

i) a, b, x must not be changed within the loop;

ii) the invariant *I* must not contain the variable x.

A further restriction, which is not explicitly mentioned in [Hoa 72], is that the given rule does not cover the case with an empty range (this was observed by Stefan Knappe [Kna 96]).

Restriction i) does also hold in some contemporary FOR-loops. In Ada e.g. x is a constant within Q and the range R is determined at the beginning in that a and b are evaluated once. a and b can be changed from within Q but this has no influence on the values in R:

```
FOR i IN a .. b          -- let a and b be integer variables
LOOP  a := 1;
      b := 0;
      r := r * i;
END LOOP;
```

The situation is similar in Pascal [ISO 7185: 53] and Fortran [ISO 1539].

In C, C++, and Java the situation is rather different: a, b, and x may be changed from within the loop and b is evaluated repeatedly [GJS 96: 280..282]. This may lead to rather bizarre behavior as it was possible in Algol 60.

Restriction ii) seems to be no restriction at all. [Hoa 72] gives no explicit reasoning for it. That it seems no real restriction will be shown in a moment after the scheme for the consistency of a specified FOR-loop has been introduced.

For the discussion of the consistency of a specified FOR-loop we use the following scheme:

```
          -- precondition  pre
FOR i IN LO..UP
LOOP  BODY
          -- invariant  inv                                    (1)
END LOOP;
          -- postcondition   post
```

The invariant inv is intended to hold after one or more executions of BODY. Therefore, it will be typically the case that inv contains the loop parameter i as e.g. in:

```
          -- pre  ≡  r = 1 AND 0 <= n AND n <= 20
FOR i IN 1 .. n
LOOP  r := r * i;
          -- inv  ≡  r = i! AND 0 <= n AND n <= 20
END LOOP;
          -- post  ≡  r = n! AND -2**63 <= r AND r <= 2**63-1
```

The invariant can also be formulated in the style of [Hoa 72] as:

$$I([1..n]) \equiv r = n! \wedge 0 <= n \wedge n <= 20 \equiv inv^{i}_{n}$$

If the proof rule of [Hoa 72] is applied we obtain in the antecedent for I([1..i]):

$$I([1..i]) \equiv I([1..n])^{n}_{i} \equiv r = i! \wedge 0 <= i \wedge i <= 20$$

which is the invariant of our loop.

As long as the loop parameter is not contained in the expressions defining the bounds a and b this approach will always fulfill restriction ii). If the loop parameter is contained in these expressions the loop may look like:

```
FOR i IN i .. i+10
LOOP  r := r + i;
END LOOP;
```

which is e.g. not allowed in Ada [ISO 8652: 5.5]; in Ada the semantics would be rather bizarre because newly declared variables are not automatically initialized. In Java a loop with such bounds is allowed [GJS 96: 82] and is well defined because i will be initialized with 0 [GJS 96: 46].

From this discussion we conclude that restriction ii) is no real restriction and that the form of the invariant in [Hoa 72] using the interval notation is unnecessarily complicated.

3 The FOR-Loop and its Ersatz Program

The FOR-loops in different programming languages differ in detailed points, e.g. whether the loop parameter (sometimes also called the controlling variable [Dah 92: 95]) is defined after termination of the loop and what its value is in this situation.

We use in this paper the FOR-loop of Ada [ISO 8652: 112] as an example of modern FOR-constructs. It has essentially the following form:

```
FOR loop_parameter IN RANGE
LOOP  BODY  END LOOP;
```

The loop_parameter is a scalar entity which is declared after the keyword FOR, i.e. it is declared locally to the FOR-loop. The loop_parameter can be used only as a *constant* inside the BODY and it cannot occur in RANGE.

The RANGE can be given in different syntactic forms. We assume for the rest of the paper the form „LO .. UP“, which is quite common. RANGE always defines a contiguous sequence of discrete values, e.g. integer numbers or elements of an enumeration type; rational numbers are not allowed. The sequence defined by RANGE may be empty. In the rest of the paper we indicate this sequence by vlo..vup (from the value of the lower bound, vlo, to the value of the upper bound, vup). vlo..vup = (vlo, succ(vlo), $succ^2$(vlo), ..., vup). If vup < vlo then the sequence is empty. The type of the loop_parameter is determined by the types of LO and UP. That these types are legal is usually checked at compile time. If LO or UP cannot be computed within the types provided by the implementation the normal execution of the FOR-loop will usually be abandoned. On the description level we will mention the constraint on the values of LO and UP by: LO, UP $\in$ TR , where TR is the value set of the type associated with the RANGE of the FOR-loop.

The BODY consists of a sequence of one or more statements. Since „NULL;“ (the empty statement) is also a legal statement BODY may be equivalent to the empty statement.

A concrete example is [KW 97: 40]:

```
FOR i IN 1 .. n
LOOP  r := r * i;  END LOOP;
```

In this case „i“ is the loop parameter. The RANGE is the sequence of integer numbers 1, 2, ..., n. If n<1 then the sequence is empty. The BODY consists of the statement „r := r*i;“.

The effect (semantics) of the FOR-loop

```
FOR i IN LO..UP                                                    (2)
LOOP  BODY  END LOOP;
```

is given by the following ersatz program (3):

```
-- create vlo and vup
CASE
   true => vlo := LO; vup := UP;
[]                                  -- nondeterministic choice     (3)
   true => vup := UP; vlo := LO;
END CASE;
IF vlo <= vup
THEN  -- let vlo..vup be  v1, v2, ..., vn, where n ≥ 1
      BODY^i_v1 ; BODY^i_v2 ; ... BODY^i_vn ;
END IF;
-- destroy vlo and vup
```

For this definition (and the rest of the paper) we assume that

(a) vlo and vup are different from all identifiers occurring in the original program,

(b) the evaluation of LO and UP does not change the program state,

(c) i is local to the FOR-loop, i.e. it cannot occur in LO or UP, and it cannot be written in BODY .

If n is statically known, the correctness of the FOR-loop can be checked using the semantics of CASE, IF, SEQ, and the elements of BODY. If n is not statically known some sort of induction proof is necessary. In the ersatz program (3) we have expressed the fact that Ada e.g. allows the evaluation of LO and UP in an arbitrary sequential order [ISO 8652: 3.5, 1.1.4]; Pascal, on the other hand, defines the order „LO, UP“ [ISO 7185: 6.8.3.9]. The ersatz program (3) has also the property that the range of values for which BODY is executed is fixed at the beginning and cannot be changed by the executions of BODY. Even if LO and UP are modified from within BODY this does not change the values assigned to vlo and vup in the CASE-statement. Since vlo and vup are different from all identifiers occurring in the original FOR-loop vlo and vup cannot be modified directly. Because of restrictions (a) and (b) the nondeterministic choice in (3) has no influence on the semantics. Therefore, we will usually not mention the nondeterministic choice in the ersatz programs in the rest of the paper.

A similar ersatz program is mentioned in [AA 78: 81]; but there it is not used as a basis for the derivation of verification rules. They use the rule of Hoare [Hoa 72] which has been discussed in sec. 2.

The FOR-loop (2) can only be executed if the declaration of vlo and vup is possible, i.e. if there is e.g. enough storage available. In this paper, we do not take this aspect into account.

4 New Proof Rules for the FOR-Loop

A proof rule or consistency condition should allow us to show {pre} FOR {post} which is equivalent to [pre ⇒ wp(FOR, post)]. If the range LO..UP is dynamic we might not be able to compute wp(FOR, post) explicitly in an efficient way. In this case, we make a proof by induction using an invariant. This proof consists of four parts: (1) proof for the empty loop, (2) connection condition at the beginning, (3) connection condition at the end, and (4) induction step for the invariant. In [Kau 98] it is shown that this implies [pre ⇒ wp(FOR, post)].

If we take

```
vlo, vup : Txy;     -- where Txy is an appropriate type
                    -- „vlo“ and „vup“ are new identifiers
vlo := LO;                                                         (4)
vup := UP;
```

```
IF vlo <= vup
THEN  -- let vlo..vup be  v1, v2, ..., vn, where n ≥ 1
      BODY^i_v1; BODY^i_v2; ... BODY^i_vn;
END IF;
```

as the definition of the semantics of the FOR-loop (2) we obtain the annotated ersatz program (5), which works also in those cases in which LO or UP are modified within BODY.

```
        -- precondition  pre
vlo, vup : Txy;                -- where Txy is an appropriate type
                                -- „vlo“ and „vup“ are new identifiers
        -- precondition  pre
vlo := LO;
vup := UP;
        -- pre ∧ vlo = LO ∧ vup = UP
IF vlo <= vup
THEN    -- let vlo..vup be  v1, v2, ..., vn, where n ≥ 1
        -- pre ∧ v1 ≤ vn ∧ v1 = LO ∧ vn = UP
      BODY^i_v1;
        -- v1 ≤ vn ∧ inv^i_v1
      BODY^i_v2;
        -- v1 ≤ vn ∧ inv^i_v2                                          (5)
        ...
      BODY^i_vn;
        -- v1 ≤ vn ∧ inv^i_vn
END IF;
        -- postcondition   post
        -- destroy vlo and vup
        -- postcondition   post
```

In (5) we use a sequential evaluation strategy for the two bounds. It is very easy to derive CCs for an Ada like evaluation strategy (as used in (3)). Since the evaluation strategy for the bounds is not essential to the semantics of the repetitive aspect of the loop we use this simpler evaluation strategy. We furthermore assume that neither declaration nor destruction of variables change the program state.

For the derivation of the CCs, (5) can be rewritten in the following stylized form (6):

```
      -- pre
ASS
      -- pre ∧ vlo = LO ∧ vup = UP
IF vlo <= vup                                                         (6)
THEN   BODY-Rep
END IF;
      -- post
```

We obtain the following consistency condition for (6):

[pre ⇒ wp(„ASS; IF;“, post)]
≡ -- sequencing rule
[pre ⇒ wp(„ASS;“, wp(„ IF;“, post))]
≡ -- IF rule
[pre ⇒ wp(„ASS;“, (vlo ≤ vup ⇒ wp(„BODY-Rep“, post)) ∧ (vlo > vup ⇒ post))]
≡ -- strong conjunctivity of wp
[pre ⇒ wp(„ASS;“, vlo ≤ vup ⇒ wp(„BODY-Rep“, post)) ∧ wp(„ASS;“, vlo > vup ⇒ post)]
≡ -- a ⇒ b ∧ c ≡ (a ⇒ b) ∧ (a ⇒ c)
[(pre ⇒ wp(„ASS;“, vlo > vup ⇒ post)) ∧
(pre ⇒ wp(„ASS;“, vlo ≤ vup ⇒ wp(„BODY-Rep“, post)))]
≡ -- vlo and vup do not occur in post; vlo does not occur in UP; wp
[(pre ⇒ LO, UP ∈ TR ∧ (LO > UP ⇒ post)) ∧

(pre ⇒ wp(„ASS;“, vlo ≤ vup ⇒ wp(„BODY-Rep“, post)))]
≡ -- vlo and vup do not occur in wp(„BODY-Rep“, post); vlo does not occur in UP; wp
[(pre ⇒ LO, UP ∈ TR ∧ (LO > UP ⇒ post)) ∧
(pre ⇒ LO, UP ∈ TR ∧ (LO ≤ UP ⇒ wp(„BODY-Rep“, post)))]
≡ -- a ⇒ b ∧ c ≡ (a ⇒ b) ∧ (a ⇒ c)
[(pre ⇒ LO, UP ∈ TR) ∧ (pre ⇒ (LO > UP ⇒ post)) ∧
(pre ⇒ LO, UP ∈ TR) ∧ (pre ⇒ (LO ≤ UP ⇒ wp(„BODY-Rep“, post)))]
≡ -- a ⇒ (b ⇒ c) ≡ a ∧ b ⇒ c
[(pre ⇒ LO, UP ∈ TR) ∧ (pre ∧ LO > UP ⇒ post) ∧ (7)
(pre ∧ LO ≤ UP ⇒ wp(„BODY-Rep“, post))]

If n is statically known (7) can possibly be computed directly (depending on BODY). In this case it is not necessary to provide an invariant. Sec. 6 contains an example for the application of (7) in this special case.

The more general case is that n is not statically known. For this case the third conjunct of (7) can be replaced by the conjunction of three conditions (8, 9, 10) which we read off from (5).

$$[\ pre \wedge v1 \le vn \wedge v1 = LO \wedge vn = UP \ \Rightarrow\ wp(„BODY^{i}_{v1};“,\ v1 \le vn \wedge inv^{i}_{v1}\ )\ ] \tag{8}$$

$$(\forall j: 1 \le j < n: [\ v1 \le vn \wedge inv^{i}_{vj} \ \Rightarrow\ wp(BODY^{i}_{v(j+1)}, v1 \le vn \wedge inv^{i}_{v(j+1)}\ )\ ]\ )$$
≡ -- [ ] is also a universal quantification (for all free variables)
$$[\ 1 \le j < n \wedge v1 \le vn \wedge inv^{i}_{vj} \ \Rightarrow\ wp(BODY^{i}_{v(j+1)}, v1 \le vn \wedge inv^{i}_{v(j+1)}\ )\ ] \tag{9}$$

$$[\ v1 \le vn \wedge inv^{i}_{vn} \ \Rightarrow\ post\ ] \tag{10}$$

At all observation points of (5) between „vup:=UP;“ and „destroy vlo and vup“ the condition

$$vlo = v1 \wedge vup = vn \tag{11}$$

holds and can be used if need arises. This condition will usually not be exploited because vlo and vup do not occur in pre, BODY and post. Because of (10) vlo and vup will usually also not occur in inv. Therefore we do not mention (11) in the CCs.

If we now replace the third conjunct in (7) by the conjunction of (8), (9) and (10) we obtain the CC (12) for the FOR-loop (2).

[(pre ⇒ LO, UP ∈ TR) ∧
(pre ∧ LO > UP ⇒ post) ∧
(pre ∧ v1 ≤ vn ∧ v1=LO ∧ vn=UP ⇒ wp(„BODY$^{i}_{v1}$;“, v1 ≤ vn ∧ inv$^{i}_{v1}$)) ∧
(1 ≤ j < n ∧ v1 ≤ vn ∧ inv$^{i}_{vj}$ ⇒ wp(BODY$^{i}_{v(j+1)}$, v1 ≤ vn ∧ inv$^{i}_{v(j+1)}$)) ∧
(v1 ≤ vn ∧ inv$^{i}_{vn}$ ⇒ post)]
≡ -- conjunctivity of wp; a ⇒ b ≡ a ⇒ a ∧ b; v1 and vn are not modified in BODY
[pre ⇒ LO, UP ∈ TR] ∧
[pre ∧ LO > UP ⇒ post] ∧
[pre ∧ v1 ≤ vn ∧ v1=LO ∧ vn=UP ⇒ wp(„BODY$^{i}_{v1}$;“, inv$^{i}_{v1}$)] ∧
[1 ≤ j < n ∧ v1 ≤ vn ∧ inv$^{i}_{vj}$ ⇒ wp(BODY$^{i}_{v(j+1)}$, inv$^{i}_{v(j+1)}$)] ∧
[v1 ≤ vn ∧ inv$^{i}_{vn}$ ⇒ post]
≡ -- elimination of superfluous variables
[pre ⇒ LO, UP ∈ TR] ∧
[pre ∧ LO > UP ⇒ post] ∧
[pre ∧ LO ≤ UP ⇒ wp(„BODY$^{i}_{v1}$;“, inv$^{i}_{v1}$)$^{v1,}_{LO,}{}^{vn,}_{UP}$] ∧ (12)
[1 ≤ j < n ∧ v1 ≤ vn ∧ inv$^{i}_{vj}$ ⇒ wp(BODY$^{i}_{v(j+1)}$, inv$^{i}_{v(j+1)}$)] ∧
[v1 ≤ vn ∧ inv$^{i}_{vn}$ ⇒ post]

(12) can now be used as a basis for CCs for specified FOR-loops of languages like Ada or Pascal. CCs for the Ada FOR-loop are the topic of sec. 5. Due to the substitution lemma for wp [Kau 98: 50] the substitution of v1 and vn in the third conjunct of (12) can only be applied after the evaluation of wp.

If LO and UP are not modified within BODY then the declaration of vlo and vup is not necessary and LO and UP can be used instead of v1 and vn. This gives us the somewhat simpler CC (13) which can be used for most of the FOR-loops used in practical program construction.

$$
\begin{array}{l}
[\ (\ pre\ \Rightarrow\ LO, UP \in TR)\ \ \wedge \\
\ (\ pre \wedge\ LO > UP\ \ \Rightarrow\ post\ )\ \ \wedge \\
\ (\ pre \wedge LO \le UP\ \ \Rightarrow\ \ wp(\text{„}BODY^{i}_{LO};\text{“},\ inv^{i}_{LO}\ )\ )\ \ \wedge \\
\ (\ 1 \le j < n\ \wedge\ LO \le UP\ \wedge\ inv^{i}_{vj}\ \Rightarrow\ \ wp(BODY^{i}_{v(j+1)},\ inv^{i}_{v(j+1)}\ )\ )\ \ \wedge \\
\ (\ LO \le UP\ \wedge\ inv^{i}_{UP}\ \ \Rightarrow\ \ post\ )\ ]
\end{array}
\qquad (13)
$$

The main difference to other proof rules for FOR-loops is that we take the evaluation of the bounds into account and that we do not require the invariant to hold before the first execution of BODY. This simplifies the invariant in certain cases and, therefore, simplifies the task of finding the invariant. The reason for this is that the invariant usually characterizes the effect of BODY, i.e. it is a sort of postcondition for BODY. This characterization must allow to establish the validity of post when $BODY^{i}_{vn}$ has been executed. A state, in which BODY has not yet been executed at all, will usually not show traces of the effect of BODY. Such a state is the state immediately before the execution of $BODY^{i}_{v1}$.

One example for this is:

```
v := 5;
    -- v = 5
FOR i IN 1 .. 10
LOOP  v := i;                                   (14)
    --  inv ???
END LOOP;
    --  v = 10
```

It is easy to see that $I1([1..i]) \equiv v = i$ is an invariant which fulfills (13).

If we apply the rule from [Hoa 72] to (14) we obtain for the invariant $I1([1..10]) \equiv v=10$, which is sufficient to establish the postcondition. We then have to determine I1([]) such that

$$(\ v=5\ \Rightarrow\ I1([])\ )\ \wedge\ (\ I1([])\ \ \Rightarrow\ wp(\text{„}v:=1;\text{“},\ v=1)\ ) \qquad (15)$$

Usually, I([]) has a form similar to I([a..b]), e.g. in a loop for summation $Is([a..b]) \equiv sum = SUM(i,i,a,b)$ [2)] and $Is([]) \equiv sum = 0$ because for $a \ge 0$ SUM(i,i,a,0) is usually defined as 0. Thus, Is([]) is derived directly from Is([a..b]).

We could try $I1([]) \equiv I1([1..10])^{10}_{pred(1)} \equiv v=0$. This does not work: $v=5 \Rightarrow v=0 \equiv$ false .

It is easy to see that $I1([]) \equiv$ true does the trick; it is maximal in that it is the weakest solution of (15). On the other hand, it is not solely derived from I1([1..i]). A more specific solution of (15) is $I1([]) \equiv v=5$, which is also not derived from I1([1..i]). In this case the solution is identical to the precondition.

An invariant which works for the rule of [Hoa 72], but is not so obvious, was found by Stefan Kauer:

$$I1'([1..i]) \equiv v=i \vee i=pred(LO) \equiv v=i \vee i=0 \qquad (16)$$

From (16) we obtain $I1'([]) \equiv I1'([1..i])^{i}_{pred(1)} \equiv v=0 \vee 0=0 \equiv$ true. This strategy does only work if pred(vlo) exists, which is not the case when vlo is the first value of a type. Even if pred(vlo) exists the invariant (16) is not entirely satisfactory: i=0 may give the impression that i - at some point in time - will assume the value 0. In Ada e.g. the type of i is „integer range 1..10“, which excludes that i assumes the value 0. (16) may do the trick, but it is questionable whether it should be used at all.

This discussion shows that things are simpler if we do not require the invariant to hold before the first execution of BODY .

[2] We use a linear notation for the summation operation: SUM(expr, index, lower bound, upper bound)

5 Proof Rules for the Ada FOR-Loop

The syntax of the Ada FOR-loop is given by:

```
FOR identifier IN [REVERSE] discrete_range                    (17)
LOOP sequence_of_statements END LOOP;
```

discrete_range may have different syntactic forms. Independent of these syntactic forms the evaluation of discrete_range yields a finite (possibly empty) totally ordered set of values. This set can be characterized by the two values vlo and vup as discussed in the preceding sections. Depending on the presence of REVERSE we obtain different ersatz programs for (17). If REVERSE is not present we obtain the ersatz program (18). For the specification of the effect we assume the same scheme (pre, inv, post) as given in (1).

```
vlo : Txy;                          -- where Txy is an appropriate type
vup : Txy;

CASE
   true => vlo := first(discrete_range);
           vup := last(discrete_range);
[]                                  -- nondeterministic choice          (18)
   true => vup := last(discrete_range);
           vlo := first(discrete_range);
END CASE;

IF vlo <= vup
THEN -- let vlo..vup be  v1, v2, ..., vn, where n ≥ 1
     BODY^i_v1 ; BODY^i_v2 ; ... BODY^i_vn ;    -- no REVERSE
END IF;

-- destroy vlo and vup
```

We use the abstract expressions „first(discrete_range)“ and „last(discrete_range)“ to indicate that the discrete_range has to be evaluated. For the CC we abbreviate these two expressions by fdr and ldr respectively. We now obtain the following CC (19) for the program (18). This CC is essentially the same as (12).

$$
\begin{aligned}
&[\ (\ pre\ \Rightarrow\ fdr, ldr \in TR)\ \wedge \\
&\ \ (\ pre \wedge\ fdr > ldr\ \ \Rightarrow\ post\ )\ \ \wedge \\
&\ \ (\ pre \wedge fdr \le ldr\ \ \Rightarrow\ \ wp(\text{„}BODY^i_{v1};\text{“},\ inv^i_{v1}\ )\ )\ \ \wedge \\
&\ \ (\ 1 \le j < n\ \wedge\ fdr \le ldr\ \wedge\ inv^i_{vj}\ \Rightarrow\ wp(BODY^i_{v(j+1)}, inv^i_{v(j+1)}\ )\ )\ \ \wedge \\
&\ \ (\ fdr \le ldr\ \wedge\ inv^i_{vn}\ \ \Rightarrow\ post\ )\ ]
\end{aligned}
\qquad (19)
$$

Under the assumptions made for this paper the nondeterministic choice of the CASE-statement has no influence on the behavior, and therefore the nondeterministic choice has no influence on the CC.

If REVERSE is present an ersatz program may look like (20).

```
vlo : Txy;                          -- where Txy is an appropriate type
vup : Txy;

CASE
   true => vlo := first(discrete_range);
           vup := last(discrete_range);
[]                                  -- nondeterministic choice          (20)
   true => vup := last(discrete_range);
           vlo := first(discrete_range);
END CASE;

IF vlo <= vup
THEN -- let vlo..vup be  v1, v2, ..., vn, where n ≥ 1
     BODY^i_vn ; ...  BODY^i_v2 ; BODY^i_v1 ;
END IF;

-- destroy vlo and vup
```

For (20) we obtain the CC (21) which reflects the reverse order of the respective executions of BODY.

$$
\begin{array}{l}
[\ (\ pre\ \Rightarrow\ fdr, ldr \in TR)\ \wedge \\
\ \ (\ pre \wedge\ fdr > ldr\ \ \Rightarrow\ post\ )\ \ \wedge \\
\ \ (\ pre \wedge fdr \le ldr\ \ \Rightarrow\ \ wp(\text{„}BODY^{i}_{vn};\text{“},\ inv^{i}_{vn}\ )\ )\ \ \wedge \\
\ \ (\ 1 < j \le n\ \wedge\ fdr \le ldr\ \wedge\ inv^{i}_{vj}\ \ \Rightarrow\ \ wp(BODY^{i}_{v(j-1)},\ inv^{i}_{v(j-1)}\ )\ )\ \ \wedge \\
\ \ (\ fdr \le ldr\ \wedge\ inv^{i}_{v1}\ \ \Rightarrow\ \ post\ )\ \ ]
\end{array}
\qquad (21)
$$

6 Examples

6.1 FOR-Loop with Statically Known Number of Repetitions

The first example contains a FOR-loop with a statically fixed range.

```
   --!pre: s=0;
FOR i in 1..10
LOOP s := s+i; END LOOP;
   --!post: s=sum(j,j,1,10);  -- sum(exp,j,1,n) = (Σj:1≤j≤n: exp)
```

This is equivalent to:

```
   --!pre: s=0;
s := s+1;
s := s+2;
s := s+3;
s := s+4;
s := s+5;
s := s+6;
s := s+7;
s := s+8;
s := s+9;
s := s+10;
   --!post: s= sum(j,j,1,10);
```

If we give this to the Frege Program Prover (FPP) [KW 97], which supports the proof of such annotated programs, we obtain the following result:

```
FPP (Frege Program Prover) University of Jena, Germany
User: 141.35.14.241       At: 1998.04.29, 15:47
The answer to your query is:
     --!pre       : (s = 0)
     --> wp       : (55 + s = 55)
     --> vc       : (s = 0 ==> 55 + s = 55)
     --> Result: proved
     s := s + 1;
     s := s + 2;
     s := s + 3;
     s := s + 4;
     s := s + 5;
     s := s + 6;
     s := s + 7;
     s := s + 8;
     s := s + 9;
     s := s + 10;
     --!post      : (s = 55)
```

Other examples of this type are the sum over the elements of a fixed array or the sum over a statically fixed sequence of input values as e.g. in [FWW 87: 70], where both the populations and the areas of the states of the USA are summed up.

6.2 FOR-Loop with Dynamic Number of Repetitions

The second example is the same as the first but the upper bound is now variable:

```
   --!pre: s=0 and 10^4≥m>0;
FOR i in 1..m
LOOP s := s+i;                                                    (22)
     --!inv: s=sum(k,k,1,i) and 0≤s≤10^9 and 10^4≥m>0
END LOOP;
   --!post: s = sum(k,k,1,m);
```

Since the new consistency condition for FOR-loops has not yet been implemented in the FPP we check this by hand. According to the bounds we may use (13), where the invariant is inv ≡ s=sum (k,k,1,i) and $0 \le s \le 10^9$ and $10^4 \ge m > 0$. In our case we have also vj=j, and we assume TR = int32 = $-2^{31}..+2^{31}-1$.

[pre ⇒ LO, UP ∈ TR]
≡ [$s=0 \wedge 10^4 \ge m>0 \Rightarrow 1, m \in$ int32]
≡ true

[pre ∧ LO > UP ⇒ post]
≡ [$s=0 \wedge 10^4 \ge m>0 \wedge 1 > m \Rightarrow s = sum(k,k,1,m)$]
≡ [false ⇒ s = sum(k,k,1,m)]
≡ true

[pre ∧ LO ≤ UP ⇒ wp(„$BODY^i_{LO}$;“, inv^i_{LO})]
≡ [$s=0 \wedge 10^4 \ge m>0 \wedge 1 \le m \Rightarrow$ wp(„s:=s+1;“, $s=sum(k,k,1,1) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0$)]
≡ [$s=0 \wedge 10^4 \ge m>0 \Rightarrow$ wp(„s:=s+1;“, $s=1 \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0$)]
≡ [$s=0 \wedge 10^4 \ge m>0 \Rightarrow s+1=1 \wedge 0 \le s+1 \le 10^9 \wedge 10^4 \ge m>0$]
≡ [$s=0 \wedge 10^4 \ge m>0 \Rightarrow s=0 \wedge 0 \le s+1 \le 10^9 \wedge 10^4 \ge m>0$]
≡ true

[$1 \le j < n \wedge$ LO ≤ UP ∧ $inv^i_{vj} \Rightarrow$ wp($BODY^i_{v(j+1)}$, $inv^i_{v(j+1)}$)]
≡ [$1 \le j < n \wedge 1 \le m \wedge s=sum(k,k,1,j) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0 \Rightarrow$
wp(„s:=s+(j+1);“, $s=sum(k,k,1,j+1) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0$)]
≡ -- in this example n = m holds; wp
[$1 \le j < m \wedge s=sum(k,k,1,j) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0 \Rightarrow$
$s+(j+1)=sum(k,k,1,j+1) \wedge 0 \le s+(j+1) \le 10^9 \wedge 10^4 \ge m>0$]
≡ -- $j \ge 1 \Rightarrow sum(k,k,1,j+1)=sum(k,k,1,j)+(j+1)$
[$1 \le j < m \wedge s=sum(k,k,1,j) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0 \Rightarrow$
$s=sum(k,k,1,j) \wedge 0 \le s+(j+1) \le 10^9 \wedge 10^4 \ge m>0$]
≡ -- $a \wedge b \Rightarrow a \wedge c \equiv a \wedge b \Rightarrow c$
[$1 \le j < m \wedge s=sum(k,k,1,j) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0 \Rightarrow 0 \le s+(j+1) \le 10^9$]
≡ -- holds also for $m=10^4 \wedge j=9\_999$ (then s=49_995_000, s+(j+1) = 50_005_000)
true

[LO ≤ UP ∧ $inv^i_{UP} \Rightarrow$ post]
≡ [$1 \le m \wedge s=sum(k,k,1,m) \wedge 0 \le s \le 10^9 \wedge 10^4 \ge m>0 \Rightarrow s=sum(k,k,1,m)$]
≡ true

All five conditions are true and, therefore, (22) is consistent.

Since the new consistency condition also covers the case UP < LO the precondition of (22) can be weakened to s=0 and $10^4 \geq m \geq -10^4$ giving us the loop (23). For the invariant we use accordingly: inv ≡ s=sum(k,k,1,i) and $0 \leq s \leq 10^9$ and $10^4 \geq m \geq -10^4$.

```
   --!pre: s=0 and 10^4≥m≥-10^4;
FOR i in 1..m                                                    (23)
LOOP s := s+i; END LOOP;
   --!post: s = sum(k,k,1,m);
```

The proof of the five conditions for the consistency of (23) is as follows.

[pre ⇒ LO, UP ∈ TR]
≡ [s=0 ∧ $10^4 \geq m \geq -10^4$ ⇒ 1, m ∈ int32]
≡ true

[pre ∧ LO > UP ⇒ post]
≡ [s=0 ∧ $10^4 \geq m \geq -10^4$ ∧ 1 > m ⇒ s = sum(k,k,1,m)]
≡ -- 1 > m ≡ sum(k,k,1,m) = 0
≡ true

[pre ∧ LO ≤ UP ⇒ wp(„$BODY^i_{LO}$;“, inv^i_{LO})]
≡ [s=0 ∧ $10^4 \geq m \geq -10^4$ ∧ 1 ≤ m ⇒
wp(„s:=s+1;“, s=sum (k,k,1,1) ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$)]
≡ [s=0 ∧ $10^4 \geq m \geq -10^4$ ∧ 1 ≤ m ⇒ wp(„s:=s+1;“, s=1 ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$)]
≡ [s=0 ∧ $10^4 \geq m \geq -10^4$ ∧ 1 ≤ m ⇒ s+1=1 ∧ $0 \leq s+1 \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[s=0 ∧ $10^4 \geq m \geq -10^4$ ∧ 1 ≤ m ⇒ s=0 ∧ $0 \leq s+1 \leq 10^9$]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[s=0 ∧ $10^4 \geq m \geq -10^4$ ∧ 1 ≤ m ⇒ $0 \leq s+1 \leq 10^9$]
≡ true

[1 ≤ j < n ∧ LO ≤ UP ∧ inv^i_{vj} ⇒ wp($BODY^i_{v(j+1)}$, $inv^i_{v(j+1)}$)]
≡ [1 ≤ j < n ∧ 1 ≤ m ∧ s=sum(k,k,1,j) and $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$ ⇒
wp(„s:=s+(j+1);“, s=sum(k,k,1,j+1) ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$)]
≡ -- in this example n = m holds
[1 ≤ j < m ∧ 1 ≤ m ∧ s=sum(k,k,1,j) ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$ ⇒
s+(j+1)=sum(k,k,1,j+1) ∧ $0 \leq s+(j+1) \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[1 ≤ j < m ∧ 1 ≤ m ∧ s=sum(k,k,1,j) ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$ ⇒
s=sum(k,k,1,j) ∧ $0 \leq s+(j+1) \leq 10^9$]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[1 ≤ j < m ∧ 1 ≤ m ∧ s=sum(k,k,1,j) ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$ ⇒ $0 \leq s+(j+1) \leq 10^9$]
≡ -- sum is monotonic; holds also for $m=10^4$ ∧ j=9_999 such that $0 \leq s+(j+1) \leq 10^9$ ≡
-- $0 \leq 50\_005\_000 \leq 10^9$ ≡ true
true

[LO ≤ UP ∧ inv^i_{UP} ⇒ post]
≡ [1 ≤ m ∧ s=sum (k,k,1,m) ∧ $0 \leq s \leq 10^9$ ∧ $10^4 \geq m \geq -10^4$ ⇒ s=sum (k,k,1,m)]
≡ true

6.3 „Exotic“ Example

Loop (24) is the somewhat exotic example mentioned already in sec. 4:

```
v := 5;
    --pre: v = 5
FOR i IN 1 .. 10
LOOP  v := i;                                                    (24)
    --inv:  v = i
END LOOP;
    --post: v = 10
```

If we substitute pre, inv and post of (24) in (13) we obtain the CC (25) for the loop (24). For TR we use again Int32.

$$
\begin{aligned}
&[\,(\,pre \Rightarrow LO, UP \in TR) \wedge \\
&\quad (\,pre \wedge LO > UP \Rightarrow post\,) \wedge \\
&\quad (\,pre \wedge LO \le UP \Rightarrow wp(\text{„}BODY^i_{LO};\text{“}, inv^i_{LO})\,) \wedge \\
&\quad (\,1 \le j < n \wedge LO \le UP \wedge inv^i_{vj} \Rightarrow wp(BODY^i_{v(j+1)}, inv^i_{v(j+1)})\,) \wedge \\
&\quad (\,LO \le UP \wedge inv^i_{UP} \Rightarrow post\,)\,] \\
\equiv\ & \\
&[\,(\,v{=}5 \Rightarrow 1, 10 \in Int32) \wedge \\
&\quad (\,v{=}5 \wedge 1 > 10 \Rightarrow v{=}10\,) \wedge \\
&\quad (\,v{=}5 \wedge 1 \le 10 \Rightarrow wp(\text{„}v{:=}1;\text{“}, v{=}1)\,) \wedge \\
&\quad (\,1 \le j < 10 \wedge 1 \le 10 \wedge v{=}j \Rightarrow wp(v{:=}j{+}1;, v{=}j{+}1)\,) \wedge \\
&\quad (\,1 \le 10 \wedge v{=}10 \Rightarrow v{=}10\,)\,] \qquad (25)\\
\equiv\ & \text{-- arithmetic, logic, wp} \\
&[\,(\,v{=}5 \Rightarrow true\,) \wedge (\,false \Rightarrow v{=}10\,) \wedge (\,v{=}5 \wedge 1 \le 10 \Rightarrow 1{=}1\,) \wedge \\
&\quad (\,1 \le j < 10 \wedge true \wedge v{=}j \Rightarrow j{+}1 = j{+}1\,) \wedge (\,true \wedge v{=}10 \Rightarrow v{=}10\,)\,] \\
\equiv\ & \text{-- logic} \\
& true
\end{aligned}
$$

6.4 Inconsistently Specified FOR-Loop

This example is very much the same as example (23) in sec. 6.2. The only difference is in the invariant which is now: inv ≡ s=sum (k,k,1,i) and $0 \le s \le 10^7$ and $10^4 \ge m \ge -10^4$. This yields the FOR-loop (26).

```
   --!pre: s=0 and 10^4≥m≥-10^4;
FOR i in 1..m
LOOP s := s+i;
     --!inv: s=sum(k,k,1,i) and 0≤s≤10^7 and 10^4≥m≥-10^4;
END LOOP;                                                        (26)
   --!post: s = sum(k,k,1,m);
```

Since only the invariant is different it suffices to prove the last three conjuncts of (13).

$$
\begin{aligned}
&[\,pre \wedge LO \le UP \Rightarrow wp(\text{„}BODY^i_{v1};\text{“}, inv^i_{v1})\,] \\
\equiv\ & [\,s{=}0 \wedge 10^4 \ge m \ge -10^4 \wedge 1 \le m \Rightarrow \\
& \qquad wp(\text{„}s{:=}s{+}1;\text{“}, s{=}sum\,(k,k,1,1) \wedge 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4)\,] \\
\equiv\ & [\,s{=}0 \wedge 10^4 \ge m \ge -10^4 \wedge 1 \le m \Rightarrow wp(\text{„}s{:=}s{+}1;\text{“}, s{=}1 \wedge 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4)\,] \\
\equiv\ & [\,s{=}0 \wedge 10^4 \ge m \ge -10^4 \wedge 1 \le m \Rightarrow s{+}1{=}1 \wedge 0 \le s{+}1 \le 10^7 \wedge 10^4 \ge m \ge -10^4\,] \\
\equiv\ & \text{-- } a \wedge b \Rightarrow a \wedge c \equiv a \wedge b \Rightarrow c \\
& [\,s{=}0 \wedge 10^4 \ge m \ge -10^4 \wedge 1 \le m \Rightarrow s{=}0 \wedge 0 \le s{+}1 \le 10^7\,] \\
\equiv\ & \text{-- } a \wedge b \Rightarrow a \wedge c \equiv a \wedge b \Rightarrow c \\
& [\,s{=}0 \wedge 10^4 \ge m \ge -10^4 \wedge 1 \le m \Rightarrow 0 \le s{+}1 \le 10^7\,] \\
\equiv\ & true
\end{aligned}
$$

$[\ 1 \le j < n\ \wedge\ LO \le UP\ \wedge\ inv^{i}_{vj}\ \Rightarrow\ wp(BODY^{i}_{v(j+1)}, inv^{i}_{v(j+1)})\ ]$
$\equiv$ -- in this example n = m holds
$[\ 1 \le j < m\ \wedge\ 1 \le m\ \wedge\ s=sum(k,k,1,j)\ \wedge\ 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4\ \Rightarrow$
$wp(„s:=s+(j+1);“, s=sum(k,k,1,j+1) \wedge\ 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4)\ ]$
$\equiv\ [\ 1 \le j < m\ \wedge\ 1 \le m\ \wedge\ s=sum(k,k,1,j) \wedge 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4\ \Rightarrow$
$s+(j+1)=sum(k,k,1,j+1) \wedge\ 0 \le s+(j+1) \le 10^7\ \wedge\ 10^4 \ge m \ge -10^4\ ]$
$\equiv$ -- $a \wedge b\ \Rightarrow\ a \wedge c\ \equiv\ a \wedge b\ \Rightarrow\ c$
$[\ 1 \le j < m\ \wedge\ s=sum(k,k,1,j)\ \wedge\ 0 \le s \le 10^7\ \wedge\ 10^4 \ge m \ge 1\ \Rightarrow\ s=sum(k,k,1,j)\ \wedge\ 0 \le s+(j+1) \le 10^7\ ]$
$\equiv$ -- $a \wedge b\ \Rightarrow\ a \wedge c\ \equiv\ a \wedge b\ \Rightarrow\ c$
$[\ 1 \le j < m\ \wedge\ s=sum(k,k,1,j)\ \wedge\ 0 \le s \le 10^7\ \wedge\ 10^4 \ge m \ge 1\ \Rightarrow\ 0 \le s+(j+1) \le 10^7\ ]$
$\equiv$ -- the negation is true
-- $(\ \exists\ j, m, s:\ 1 \le j < m\ \wedge\ s=sum(k,k,1,j)\ \wedge\ 0 \le s \le 10^7\ \wedge\ 10^4 \ge m \ge 1\ \wedge$
-- $\neg\ (\ 0 \le s+(j+1) \le 10^7\ )\ )$
-- $\equiv$ -- m = 5000, j = 4471, s = 9_997_156 $\le 10^7$, s+(j+1) = 10_001_628 $> 10^7$
-- true
false

$[\ LO \le UP\ \wedge\ inv^{i}_{UP}\ \Rightarrow\ post\ ]$
$\equiv\ [\ 1 \le m\ \wedge\ s=sum\ (k,k,1,m)\ \wedge\ 0 \le s \le 10^7\ \wedge 10^4 \ge m \ge -10^4\ \Rightarrow\ s=sum\ (k,k,1,m)\ ]$
$\equiv$ true

Since one conjunct is false (26) is not consistent. We observe that the erroneous values are not the maximal values for m and j. The idea to try to check the condition by using the maximal values for m and j is fallacious:

$1 \le j < m\ \wedge\ 1 \le m\ \wedge\ s=sum(k,k,1,j)\ \wedge\ 0 \le s \le 10^7\ \wedge\ 10^4 \ge m \ge -10^4\ \Rightarrow\ 0 \le s+(j+1) \le 10^7$
$\equiv$ -- $m = 10^4$, j = 9_999, s = 49_995_000 $> 10^7$, s+(j+1) = 50_005_000 $> 10^7$ (27)
false $\Rightarrow$ false
$\equiv$ true

The erroneous value for j gives exactly that value of the loop parameter i for which the execution of BODY fails for the first time: i = j+1 = 4472. In typical implementations the normal execution of the loop would then be abandoned. As a consequence the situation of (27) would never arise.

6.5 FOR-Loop in which BODY Modifies Bounds

This example (28) is very much the same as example (22) in sec. 6.2. The only difference is that BODY sets the value of m to zero. According to the assumptions of sec. 3 this should not change the behavior wrt s.

```
   --!pre: s=0 and 10^4≥m>0 and m=Km;
FOR i in 1..m
LOOP m := 0;
     s := s+i;                                                    (28)
     --!inv: s=sum(k,k,1,i) and 0≤s≤10^9 and 10^4≥Km>0;
END LOOP;
   --!post: s = sum(k,k,1,Km);
```

Since the bounds are modified within BODY we have to use CC (12).

$[\ pre\ \Rightarrow\ LO, UP \in TR\ ]$
$\equiv\ [\ s=0\ \wedge\ 10^4 \ge m > 0\ \wedge\ m=Km\ \Rightarrow\ 1, m \in Int32\ ]$
$\equiv$ true

[pre ∧ LO > UP ⇒ post]
≡ [s=0 ∧ 10^4≥m>0 ∧ m=Km ∧ 1 > m ⇒ s = sum(k,k,1,Km)]
≡ [false ⇒ s = sum(k,k,1,Km)]
≡ true

[pre ∧ LO ≤ UP ⇒ wp(„$BODY^i_{v1}$“, inv^i_{v1})$^{v1,\ vn,}_{LO,\ UP}$]
≡ [s=0 ∧ 10^4≥m>0 ∧ m=Km ∧ 1 ≤ m ⇒
wp(„(m:=0; s:=s+i;)$^i_1$“, (s=sum (k,k,1,i) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)$^i_1$)$^{v1,\ vn,}_{1,\ m}$]
≡ [s=0 ∧ 10^4≥m>0 ∧ m=Km ∧ 1 ≤ m ⇒
wp(„m:=0; s:=s+1;“, s=sum (k,k,1,1) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)$^{v1,\ vn,}_{1,\ m}$]
≡ -- wp
[s=0 ∧ 10^4≥m>0 ∧ m=Km ⇒
(s+1=sum (k,k,1,1) ∧ 0≤s+1≤10^9 ∧ 10^4≥Km>0)$^{v1,\ vn,}_{1,\ m}$]
≡ [s=0 ∧ 10^4≥m>0 ∧ m=Km ⇒ s+1=1 ∧ 0≤s+1≤10^9 ∧ 10^4≥Km>0]
≡ -- eliminate Km
[s=0 ∧ 10^4≥m>0 ⇒ s=0 ∧ 0≤s+1≤10^9 ∧ 10^4≥m>0]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[s=0 ∧ 10^4≥m>0 ⇒ 0≤s+1≤10^9]
≡ true

[1 ≤ j < n ∧ v1 ≤ vn ∧ inv^i_{vj} ⇒ wp($BODY^i_{v(j+1)}$, $inv^i_{v(j+1)}$)]
≡ -- v1=1 ∧ vn=Km
[1 ≤ j < n ∧ 1 ≤ Km ∧ (s=sum (k,k,1,i) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)^i_j ⇒
wp((„m:=0; s:=s+i;“)$^i_{j+1}$, (s=sum (k,k,1,i) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)$^i_{j+1}$))]
≡ -- n = Km
[1 ≤ j < Km ∧ 1 ≤ Km ∧ s=sum (k,k,1,j) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒
wp(„m:=0; s:=s+(j+1);“, s=sum (k,k,1,j+1) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)]
≡ -- wp
[1 ≤ j < Km ∧ s=sum (k,k,1,j) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒
s+(j+1)=sum (k,k,1,j+1) ∧ 0≤s+(j+1)≤10^9 ∧ 10^4≥Km>0]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c; j≥1 ⇒ sum(k,k,1,j+1)=sum(k,k,1,j)+(j+1)
[1 ≤ j < Km ∧ s=sum (k,k,1,j) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒
s+(j+1)=sum (k,k,1,j)+(j+1) ∧ 0≤s+(j+1)≤10^9]
≡ [1 ≤ j < Km ∧ s=sum (k,k,1,j) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒ s=sum (k,k,1,j) ∧ 0≤s+(j+1)≤10^9]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[1 ≤ j < Km ∧ s=sum (k,k,1,j) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒ 0≤s+(j+1)≤10^9]
≡ -- holds also for Km=10^4 ∧ j=9_999 (then s=49_995_000, s+(j+1) = 50_005_000)
true

[v1 ≤ vn ∧ inv^i_{vn} ⇒ post]
≡ [1 ≤ Km ∧ (s=sum (k,k,1,i) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)$^i_{Km}$ ⇒ s = sum(k,k,1,Km)]
≡ [1 ≤ Km ∧ s=sum (k,k,1,Km) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒ s = sum(k,k,1,Km)]
≡ -- a ∧ b ⇒ a
true

Since all five clauses are true (28) is consistent.

We observe that in this case the fifth clause of (13) would give a wrong result.

[(LO ≤ UP ∧ inv^i_{UP} ⇒ post)]
≡ [(1 ≤ m ∧ (s=sum (k,k,1,i) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0)$^i_{UP}$ ⇒ s = sum(k,k,1,Km)]
≡ -- UP = m
[1 ≤ m ∧ s=sum (k,k,1,m) ∧ 0≤s≤10^9 ∧ 10^4≥Km>0 ⇒ s = sum(k,k,1,Km)]
≡ -- m=1, Km=2, s=1

$(1 \le 1 \wedge 1=\text{sum}(k,k,1,1) \wedge 0 \le 1 \le 10^9 \wedge 10^4 \ge 2 > 0 \Rightarrow 1 = \text{sum}(k,k,1,2)) \wedge Q$
$\equiv (1 \le 1 \wedge 1=1 \wedge 0 \le 1 \le 10^9 \wedge 10^4 \ge 2 > 0 \Rightarrow 1 = 2) \wedge Q$
$\equiv (\text{true} \Rightarrow \text{false}) \wedge Q$
$\equiv \text{false} \wedge Q$
$\equiv \text{false}$

7 FOR-Loop as a special WHILE-Loop

As already mentioned in sec. 2 the FOR-loop is treated quite rarely in the literature on program verification. If it is mentioned at all, it is often defined using an ersatz program based on the WHILE-loop [AA 78: 81; Ams 87: 88, 89] Similarly, the Pascal standard defines the semantics of the FOR-loop by an ersatz program based on the WHILE-loop [ISO 7185: 6.8.3.9]. Zuse also reduces the FOR-loop to a WHILE-loop [Zus 72: 4-34]. For the FOR-loop

```
FOR i := a TO b DO S                                    (29)
```

[AA 78: 81] uses the ersatz program (30).

```
IF a ≤ b THEN
BEGIN i := a; S;
      WHILE i < b DO
      BEGIN i := succ(i);                               (30)
            S;
      END
END
```

Sometimes, a more naïve ersatz program (31) is proposed [Ams 87: 88, 89; App 97: 163].

```
LET VAR i := a
    VAR limit := b
IN WHILE i ≤ limit                                      (31)
   DO ( S;  i := i+1 )
END
```

Program (31) does not work correctly if b = last (type (i)), a fact which is also mentioned by Appel and by Dahl [Dah 92: 95]. This means that (31) is not really an ersatz program for (29).

In (30) a is evaluated once or twice and b is evaluated (n+1) times, where n is the number of executions of S. These repeated evaluations are avoided in the ersatz program given in the Pascal standard. Because of the repeated evaluations of b (30) is in general not equivalent to (29). If b is modified from within S the behavior may be different. If we combine (30) and (31) we obtain ersatz program (32), which is nearly the same as in the Pascal standard. Apart from the nondeterministic evaluation of a and b (32) leads to essentially the same computation as (3). Therefore, (32) is an ersatz program for the FOR-loop (29) according to the concept of FOR-loop in this paper. (30) would be more appropriate for more dynamic FOR-loops as e.g. in Algol 60 or Java.

```
BEGIN temp1 := a;
      temp2 := b;
   IF temp1 ≤ temp2
   THEN  i := temp1;
         S;                                             (32)
         WHILE i < temp2
         DO  i := succ(i);  S;
         END;
   END;
END;
```

The difference between (32) and the ersatz program given in the Pascal standard is that Pascal uses a different condition in the WHILE-loop: „WHILE i <> temp2“. Dahl also gives a correct ersatz program based on

a LOOP-WHILE-REPEAT-loop in which the WHILE clause appears after the loop body S. This avoids the duplication of S in the ersatz program [Dah 92: 95]

A proof of [pre ⇒ wp((32), post)] involves the computation of wp(WHILE, post). If the number of repetitions of the WHILE-loop is statically known this can be done as shown in the example in sec. 6.1. If the number of repetitions is not statically known we can apply the scheme based on invariant and termination function.[3] The annotated program (33) uses the invariant inv and the termination function tf, whose range is the set of positive integers. T is a fresh identifier of type Natural. For the sake of simplicity we assume that the type of a, b and i is of kind Integer. Usually, general discrete types are allowed for the range of a FOR-loop. But most languages provide mappings from discrete types to Integer, as e.g. the function Ord in Pascal or the attribute Pos in Ada, and therefore, this is no serious restriction. In (33) we use the same names as in (3) in order to make the comparison between the CCs easier.

```
    -- pre ∧ v1=LO ∧ vn=UP
BEGIN    -- LO, UP ∈ TR
      vlo, vup, i : Txy;
      vlo := LO;
      vup := UP;
   IF vlo ≤ vup
   THEN  -- vlo=LO ∧ vup=UP ∧ v1=LO ∧ vn=UP ∧ v1 ≤ vn
      i := vlo;
         -- vlo=LO ∧ vup=UP ∧ v1=LO ∧ vn=UP ∧ v1 ≤ vn ∧ i=vlo
      BODY;
         -- pre-while                                                    (33)
         -- inv ∧ tf ≥ 0 ∧ i=v1 ∧ vlo=v1 ∧ vup=vn ∧ v1 ≤ vn ∧ i ≤ vn
      WHILE i < vup
      LOOP   -- inv ∧ tf > 0 ∧ T = tf ∧ i<vn ∧ vlo=v1 ∧ vup=vn ∧ v1<vn
         i := i+1;  BODY;
             -- inv ∧ tf ≥ 0 ∧ tf < T ∧ i≤vn ∧ vlo=v1 ∧ vup=vn ∧ v1<vn
      END LOOP;
         -- i ≥ vn ∧ (inv ∧ tf ≥ 0 ∧ i=v1 ∧ vlo=v1 ∧ vup=vn ∧ v1=vn ∧ i=vn ∨
                     (inv ∧ tf ≥ 0 ∧ tf < T ∧ i≤vn ∧ vlo=v1 ∧ vup=vn ∧ v1<vn)
         -- post-while
   ELSE NULL;
   END IF;
END BEGIN;
   -- destroy vlo, vup, i
   -- post
```

Apart from pre, post and inv we have three additional elements in (33): the assertions pre-while and post-while, and the termination function tf. For tf we can always use „vn - i“. For post-while the best we can use is post because the program state is the same after END LOOP, after END IF and after END BEGIN. For pre-while the best we can use is

sp(vlo≤vup ∧ sp(pre ∧ v1=LO ∧ vn=UP, „vlo:=LO; vup:=UP;“), „i:=vlo; BODY;“)

where sp means „strongest postcondition“.

For this condition we obtain:

sp(vlo≤vup ∧ sp(pre ∧ v1=LO ∧ vn=UP, „vlo:=LO; vup:=UP;“), „i:=vlo; BODY;“)
≡ -- vlo, vup and i do not occur in pre, LO and UP
sp(vlo≤vup ∧ pre ∧ v1=LO ∧ vn=UP ∧ vlo=LO ∧ vup=UP , „i:=vlo; BODY;“)
≡ -- sp
sp(sp(vlo≤vup ∧ pre ∧ v1=LO ∧ vn=UP ∧ vlo=LO ∧ vup=UP, „i:=vlo;“), „BODY;“)
≡ -- i does not occur in vlo, vup, pre, LO and UP
sp(vlo≤vup ∧ pre ∧ v1=LO ∧ vn=UP ∧ vlo=LO ∧ vup=UP ∧ i=vlo, „BODY;“)

[3] In special cases it is possible to compute wp(WHILE, post) using the method of Kauer [Kau 98].

If we apply these observations to (33) we obtain the annotated program (34). As in sec. 4 we do no longer mention vlo and vup in the assertions and CCs but use v1 and vn instead.

```
    -- pre ∧ v1=LO ∧ vn=UP
BEGIN    -- LO, UP ∈ TR
      vlo, vup, i : Txy;
      vlo := LO;
      vup := UP;
         -- pre ∧ v1=LO ∧ vn=UP
   IF vlo ≤ vup
   THEN  -- pre ∧ v1=LO ∧ vn=UP ∧ v1 ≤ vn
      i := vlo;
         -- pre ∧ v1=LO ∧ vn=UP ∧ v1 ≤ vn ∧ i=v1
      BODY;
         -- sp(pre ∧ v1=LO ∧ vn=UP ∧ v1 ≤ vn ∧ i=v1,  „BODY;“ )                (34)
         -- inv ∧ vn-i ≥ 0 ∧ i=v1 ∧ v1 ≤ vn ∧ i ≤ vn
      WHILE i < vup
      LOOP   -- inv ∧ vn-i > 0 ∧ T = vn-i ∧ i<vn ∧ v1<vn
         i := i+1;  BODY;
             -- inv ∧ vn-i ≥ 0 ∧ vn-i < T ∧ i≤vn ∧ v1<vn
      END LOOP;
         -- i ≥ vn ∧ (inv ∧ vn-i ≥ 0 ∧ i=v1 ∧ v1=vn ∧ i=vn ∨
                     (inv ∧ vn-i ≥ 0 ∧ vn-i < T ∧ i≤vn ∧ v1<vn)
         -- post
   ELSE
        -- pre ∧ v1=LO ∧ vn=UP ∧ v1 > vn
     NULL;
        -- post
   END IF;
END BEGIN;
   -- destroy vlo, vup, i
   -- post
```

Some assertions in (34) can be further simplified, leading to the annotated ersatz program (35)

```
    -- pre ∧ v1=LO ∧ vn=UP
BEGIN    -- LO, UP ∈ TR
      vlo, vup, i : Txy;
      vlo := LO;
      vup := UP;
         -- pre ∧ v1=LO ∧ vn=UP
   IF vlo ≤ vup
   THEN  -- pre ∧ v1=LO ∧ vn=UP ∧ v1≤vn
      i := vlo;
         -- pre ∧ v1=LO ∧ vn=UP ∧ v1≤vn ∧ i=v1
      BODY;
         -- sp(pre ∧ v1=LO ∧ vn=UP ∧ v1≤vn ∧ i=v1,  „BODY;“ )  -- (*)          (35)
         -- inv ∧ i=v1 ∧ v1≤vn ∧ i≤vn
      WHILE i < vup
      LOOP   -- inv ∧ T=vn-i ∧ i<vn ∧ v1<vn
         i := i+1;  BODY;
             -- inv ∧ vn-i<T ∧ i≤vn ∧ v1<vn
      END LOOP;
         -- i=vn ∧ inv ∧ (v1=vn ∨ v1<vn ∧ vn-i<T)
         -- post
   ELSE
        -- pre ∧ v1=LO ∧ vn=UP ∧ v1>vn
     NULL;
        -- post
   END IF;
END BEGIN;
   -- destroy vlo, vup, i
```

```
-- post
```

Most of the CC can be read off directly from (35). For the induction step we obtain

[inv ∧ T = vn-i ∧ i<vn ∧ v1<vn ⇒ wp(„i := i+1; BODY;“, inv ∧ vn-i < T ∧ i≤vn ∧ v1<vn)]
≡ -- v1, vn, i and T are not modified within BODY, strong conjunctivity of wp
[inv ∧ T = vn-i ∧ i<vn ∧ v1<vn ⇒ vn-i-1 < T ∧ i+1≤vn ∧ v1<vn ∧ wp(„i := i+1; BODY;“, inv)]
≡ -- T does not occur in BODY and inv; eliminate T
[inv ∧ i<vn ∧ v1<vn ⇒ (vn-i)-1 < vn-i ∧ i+1≤vn ∧ v1<vn ∧ wp(„i := i+1; BODY;“, inv)]
≡ -- arithmetic; a ⇒ b ≡ a ⇒ a ∧ b
[inv ∧ i<vn ∧ v1<vn ⇒ wp(„i := i+1; BODY;“, inv)]

Using this we obtain the complete CC (36).

[pre ⇒ LO, UP ∈ TR] ∧
[pre ∧ LO > UP ⇒ post] ∧ (36)
[sp(pre ∧ v1=LO ∧ vn=UP ∧ v1 ≤ vn ∧ i=v1, „BODY;“) ⇒ inv ∧ i=v1 ∧ v1 ≤ vn ∧ i ≤ vn] ∧
[inv ∧ i<vn ∧ v1<vn ⇒ wp(„i := i+1; BODY;“, inv)] ∧
[i=vn ∧ inv ∧ (v1=vn ∨ v1<vn ∧ vn-i < T) ⇒ post]

(36) corresponds to (12) but looks somewhat more complicated.

Again, as in sec. 4 the CC can be simplified if LO and UP are not modified from within S giving us the CC (37).

[pre ⇒ LO, UP ∈ TR] ∧
[pre ∧ LO > UP ⇒ post] ∧
[sp(pre ∧ LO ≤ UP ∧ i=LO, „BODY;“) ⇒ inv ∧ i=LO ∧ LO ≤ UP ∧ i ≤ UP] ∧ (37)
[inv ∧ i<UP ∧ LO<UP ⇒ wp(„i := i+1; BODY;“, inv)] ∧
[i=UP ∧ inv ∧ (LO=UP ∨ LO<UP ∧ UP-i < T) ⇒ post]

If we use (37) with the inconsistently specified FOR-loop (26) we obtain the result below. As invariant we use inv ≡ s=sum (k,k,1,i) and $0 \le s \le 10^7$ and $10^4 \ge m \ge -10^4$ and i≥1 . The first two clauses in (37) are the same as the first two in (13). They have already been proved in sec. 6.2. We therefore only need to prove the remaining three clauses of (37).

[sp(pre ∧ LO ≤ UP ∧ i=LO, „BODY;“) ⇒ inv ∧ i=LO ∧ LO ≤ UP ∧ i ≤ UP]
≡ [sp(s=0 ∧ $10^4 \ge m \ge -10^4$ ∧ 1 ≤ m ∧ i=1, „s:=s+i;“) ⇒
s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m \ge -10^4$ ∧ i≥1 ∧ i=1 ∧ 1 ≤ m ∧ i ≤ m]
≡ -- sp
[s=1 ∧ $10^4 \ge m \ge 1$ ∧ i=1 ⇒ s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m \ge 1$ ∧ i=1 ∧ i ≤ m]
≡ -- a ∧ b ⇒ a ∧ c ≡ a ∧ b ⇒ c
[s=1 ∧ $10^4 \ge m \ge 1$ ∧ i=1 ⇒ s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ i ≤ m]
≡ true

[inv ∧ i<UP ∧ LO<UP ⇒ wp(„i := i+1; BODY;“, inv)] ∧
≡ [s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m \ge -10^4$ ∧ i≥1 ∧ i<m ∧ 1<m ⇒
wp(„i := i+1; s:=s+i;“, s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m \ge -10^4$ ∧ i≥1)]
≡ [s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ i≥1 ∧ i<m ⇒
wp(„i := i+1; s:=s+i;“, s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ i≥1)]
≡ -- wp
[s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ i≥1 ∧ i<m ⇒
wp(„i := i+1;“, s+i=sum (k,k,1,i) ∧ $0 \le s+i \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ i≥1)]

≡ -- wp
[s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ $i \ge 1$ ∧ $i < m$ ⇒
$s+(i+1)=sum(k,k,1,i+1)$ ∧ $0 \le s+(i+1) \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ $i+1 \ge 1$]

≡ [s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ $i \ge 1$ ∧ $i < m$ ⇒
$s=sum(k,k,1,i)$ ∧ $0 \le s+(i+1) \le 10^7$ ∧ $i \ge 0$]

≡ -- $a \wedge b \Rightarrow a \wedge c \equiv a \wedge b \Rightarrow c$; $i \ge 1 \wedge i \ge 0 \equiv i \ge 1$
[s=sum (k,k,1,i) ∧ $0 \le s \le 10^7$ ∧ $10^4 \ge m > 1$ ∧ $i \ge 1$ ∧ $i < m$ ⇒ $0 \le s+(i+1) \le 10^7$]

≡ -- the same condition as in sec. 6.4
false

[$i = UP \wedge inv \wedge (LO = UP \vee LO < UP \wedge UP - i < T) \Rightarrow post$]
≡ [$i = m \wedge s = sum(k,k,1,i) \wedge 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4 \wedge i \ge 1 \wedge (1 = m \vee 1 < m \wedge m - i < T) \Rightarrow$
$s = sum(k,k,1,m)$]
≡ -- eliminate i
≡ [$s = sum(k,k,1,m) \wedge 0 \le s \le 10^7 \wedge 10^4 \ge m \ge -10^4 \wedge m \ge 1 \wedge (1 = m \vee 1 < m \wedge m - m < T) \Rightarrow$
$s = sum(k,k,1,m)$]
≡ -- true

We obtain essentially the same result as in sec. 6.4 but the derivation is somewhat more complicated.

8 Conclusions

In this paper we have developed new proof rules for FOR-loops. These rules are appropriate for FOR-loops in existing programming languages as e.g. Ada or Pascal and therefore, may be readily applied to programs in those languages. In the literature we often find rules which are based on some idealized form of FOR-loops. A second improvement is that the new rules do not require the invariant to hold before the first execution of the loop body and that they also cover the case in the which the loop body is not executed at all. It still remains to develop proof rules for more complicated FOR-loops as e.g. those of Java or C++.

Acknowledgment

I am grateful to M. Hartmeier and S. Kauer for useful hints to earlier drafts of this paper. F. Ecke gave very useful hints which improved my English.

References

AA 78 Alagic, Suad; Arbib, Michael A.: The Design of Well-Structured and Correct Programs. Springer, New York etc., 1978. 0-387-90299-6

Abr 96 Abrial, J.-R.: The B-Book - Assigning Programs to Meanings. Cambridge University Press, 1996. 0-521-49619-5

Ams 87 Amstel, J. J. van: Die Entwicklung von Algorithmen in Pascal. Addison-Wesley, Bonn etc, 1987. 3-925118-19-5

AO 94 Apt, Krysztof R.; Olderog, Ernst-Rüdiger: Programmverifikation. Springer, Berlin etc., 1994. 3-540-57479-4

App 97 Appel, Andrew W.: Modern Compiler Implementation in Java. Cambridge University Press, Cambridge, 1997. 0-521-58387-X

Bab 87 Baber, Robert Laurence: The Spine of Software. John Wiley & Sons, Chichester etc., 1987. 0-471-91474-6

Bab 91 Baber, Robert Laurence: Error Free Software. John Wiley & Sons, Chichester etc., 1991. 0-471-93016-4

Bac 81 Backus, John: The History of FORTRAN I, II, and III. = [Wex 81: 25 .. 45]

BBG 63 Backus, J.W.; Bauer, F.L.; Green, J.; Katz, C.; McCarthy, J.; Naur, P.; Perlis, A.J.; Rutishauser, H.; Samelson, K.; Vauquois, B.; Wegstein, J.H.; Wijngaarden, A. van; Woodger, M.: Revised Report on the Algorithmic Language ALGOL 60. CACM 6, 1 (1963) 1 .. 17

Bes 95 Best, Eike: Semantik. Friedr. Vieweg & Sohn, Braunschweig, 1995. 3-528-05431-X. (also in English: 0-13-460643-4)

BK 94 Beitz, W.; Küttner, K.-H. (eds.): DUBBEL - Handbook of Mechanical Engineering. Springer, Berlin etc. 1994. 3-540-19868-7

Bow 55 Bowden, B. V. (ed.): Faster Than Thought. Sir Isaac Pitman & Sons, London, 1955

Cou 92 Cousot, Patrick: Methods and Logics for Proving Programs. = Leeuwen, Jan van: Handbook of Theoretical Computer Science, Volume B, Formal Models and Semantics. Elsevier / MIT Press, Amsterdam etc. / Cambridge, 2nd pr. 1992, 841..993. 0-444-88074-7

Dah 92 Dahl, Ole-Johan: Verifiable Programming. Prentice Hall, New York etc., 1992. 0-13-951062-1

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

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

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

Flo 67 Floyd, Robert W.: Assigning Meanings to Programs. AMS Proc. of Symposia in applied Mathematics, Vol. 19, Am. Math. Soc., Providence, Rhode Island, 1967, 19..32

Fra 92 Francez, Nissim: Program Verification. Addison-Wesley, Wokingham etc., 1992. 0-201-41608-5

Fut 89 Futschek, Gerald: Programmentwicklung und Verifikation. Springer, New York etc., 1989. 0-387-81867-7

FWW 87 Watt, David A.; Wichmann, Brian A.; Findlay, William: ADA - Language and Methodology. Prentice/Hall Int., Englewood Cliffs etc., 1987. 0-13-004078-9

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

Gri 83 Gries, David: The Science of Programming. Springer, New York. etc., 1983. 0-387-90641-X

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

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

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

HW 73 Hoare, C.A.R.; Wirth, N.: An Axiomatic Definition of the Programming Language Pascal. Acta Informatica 2,4 (1973) 335...355

ISO 1539 ISO/IEC 1539:1991: Information technology -- Programming languages -- FORTRAN. ISO / IEC, Genève, 1991

ISO 7185 ISO / IEC 7185:1990(E): Information Technology - Programming Languages - Pascal. ISO / IEC, Genève, 1990

ISO 8652 ISO / IEC 8652:1995(E): Information Technology - Programming Languages - Ada. ISO / IEC, Genève, 1995

Kau 98 Kauer, Stefan: Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. Dissertation, Friedrich Schiller University, 1998

Kna 96 Knappe, Stefan: Computation of the Verification Conditions for a Subset of Ada. Diploma Thesis, Friedrich Schiller University, Dept. of Computer Science, Jena, 1996.May.02. (in German)

KW 97 Winkler, Jürgen F. H.; Kauer, Stefan: Proving Assertions is also Useful. SIGPLAN Notices 32,3 (1997) 38..41

McC 81 McCarthy, John: History of LISP. = [Wex 81: 173 .. 185]

Men 55 Menabrea, L. F.: Sketch of the Analytical Engine invented by Charles Babbage, Esq. = [Bow 55: 341..408] = Reprint from Taylor's Scientific Memoirs, Vol. III, 1843, 666..731

PR 97 Rechenberg, Peter; Pomberger, Gustav (eds.): Informatik-Handbuch. Carl Hanser, München usw. 1997. 3-446-18691-3

Sam 69 Sammet, Jean E.: Programming Languages: History and Fundamentals. Prentice Hall, Englewood Cliffs, 1969

Tuc 97 Tucker, Allen B. Jr. (ed.): The Computer Science and Engineering Handbook. CRC Press, Boca Raton, 1997. 0-8493-2909-4

Wex 81 Wexelblat, Richard L. (ed.): History of Programming Languages. Academic Press, New York etc., 1981. 0-12-745040-8

Win 96 Winkler, Jürgen F. H.: Some Properties of the Smallest Post-Set and the Largest Pre-Set of Abstract Programs. Friedrich Schiller University, Jena, Dept. of Math. and Computer Science, Report Math / Inf / 96 / 32, http://www1.informatik.uni-jena.de/themen/pap-talk/wp-stat5.ps

Zus 72 Zuse, Konrad: Der Plankalkül. Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Bericht BMBW-GMD-63, 1972

Zus 89 Zuse, Konrad: The Plankalkül. Oldenbourg, München, Wien, 1989. 3-486-21288-5