Journal of Symbolic Computation 45 (2010) 1101–1113

Contents lists available at ScienceDirect

Journal of Symbolic Computation

journal homepage: www.elsevier.com/locate/jsc

Mechanical inference of invariants for FOR-loops

Stefan Kauer [a], Jürgen F.H. Winkler [b,1]

[a] EADS Deutschland GmbH, Business Unit Defence Electronics, Claude-Dornier-Str. D-88090 Immenstaad, Germany
[b] Friedrich Schiller University, Institute of Informatics, Ernst-Abbe-Platz 2, D-07743 Jena, Germany

ARTICLE INFO

Article history:
Received 1 July 2008
Accepted 10 November 2008
Available online 9 March 2009

Keywords:
Mechanical verification
Mechanical inference of loop invariants
FOR-loop
RCPV

ABSTRACT

In the mechanical verification of programs containing loops it is often necessary to provide loop invariants additionally to the specification in the form of pre and postconditions. In this paper we present a method for the mechanical inference of invariants for a practically relevant class of FOR-loops. The invariant is derived from the specification (pre, post) and the final bound of the loop only. The method is based on the technique "replacing a constant in post by a variable", which has traditionally been used manually for the development of WHILE-loops. Our method is a complete mechanization of this heuristic for the verification of existing annotated FOR-loops. The range of applicability of the method is further extended by a technique called "bound transformation" and by taking common invariant conjuncts of pre and post into account. As a result, the method is applicable to the majority of FOR-loops occurring in practice.

The incorporation of this method into an automatic program verifier would make the task of the SW engineer easier, because he has only to provide a pre–post-specification for a FOR-loop.

1. Introduction

Program verification involves a great amount of mechanical formula manipulation. If done by hand, this is tedious and, even worse, error prone. Most of the theorems (verification conditions (VC)), which have to be proved, are quite trivial and can therefore be proved automatically by an automatic theorem prover. If all VCs are generated and proved automatically we speak of automatic program verification.

E-mail addresses: Stefan.Kauer@eads.com (S. Kauer), jwinkler@acm.org, winkler@informatik.uni-jena.de (J.F.H. Winkler).
[1] Tel.: +49 3641 9 46340; fax: +49 3641 9 46302.

0747-7171/$ – see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jsc.2008.11.008

A tool which performs automatic program verification is then called an automatic program verifier (APV). Examples of such tools are Boogie (Barnett et al., 2006), FPP (Kauer and Winkler, 1999; Winkler, 1997) and NPPV (Gumm, 1999). Other tools use a combination of automatic and interactive theorem proving and can therefore be called semi-automatic program verifiers. Examples are KeY (Ahrendt et al., 2005), SPARK (Barnes, 2000) and Theorema (Kovács et al., 2003).

Most tools for the verification of concrete programs use the assertion based method (ABM) for the specification of the required behavior of the program (e.g. Boogie, FPP, KeY, NPPV, SPARK, Theorema). The specification is given by a pair (pre, post) of assertions which refer to entities of the program, and may also refer to entities which belong to the specification only. ABM allows also the verification of program fragments and therefore can be used by the SW engineer in a continuous manner during program development, and not only for the verification of a finished program in one big step. In this situation, the use of an APV is especially convenient.

In ABM automatic verification on the basis of (pre, post) is rather straightforward for statements like declarations, assignment, IF, and CASE.[2] The verification of loops usually requires also an invariant (Dijkstra, 1976; Hoare, 1972; Leavens et al., 2008; Turing, 1949; Winkler, 1998), and for WHILE-loops additionally a termination function (Dijkstra, 1976; Floyd, 1967; Leavens et al., 2008; Turing, 1949). It would be easier if loops could also be verified by giving only (pre, post). This can be done in two ways: (1) by computing wp(loop, post) resp. sp(pre, loop) and use this in the general verification condition pre $\Rightarrow$ wp(loop, post) resp. sp(pre, loop) $\Rightarrow$ post, or (2) by computing an invariant (and in the case of a WHILE-loop a termination function) and then perform the verification using approximative VCs, as e.g. (5). Automatic computation of invariants from the code is seen as difficult in the general case (Back, 2006). Stefan Kauer has developed methods for the mechanical verification of certain loops, where it is sufficient to provide only precondition and postcondition, but no invariant (Kauer, 1999). For WHILE-loops his method computes wp(loop, post). For FOR-loops his method is based on the heuristic "replacing a constant in the postcondition by a variable (RCPV)" for the computation of an invariant. For the more complicated cases this method uses a second FOR-loop which is derived from the original one (Kauer and Winkler, 2007). In this paper we present an improvement of Kauer and Winkler (2007), which avoids the use of a derived loop and is applicable in more cases than the old method. For the verification of FOR-loops we use the VC of Winkler (1998) (see (5)) which is less restrictive than the proof rule of Hoare (1972).

An annotated FOR-loop (AFL) in Ada syntax looks like

```
   -- PRE
FOR i in LO..UP LOOP  BODY  END LOOP;                    (1)
   -- POST
```

where i is the loop variable, the value of LO is the lower bound and the value of UP is the upper bound of the loop. (1) is an upwards counting loop. Many languages contain also downwards counting loops. In this paper we deal mainly with upwards counting FOR-loops. Downwards counting FOR-loops do not pose new problems and can be treated in an analogous manner (Kauer and Winkler, 2000; Winkler, 1998).

The FOR-loop (1) is a strictly controlled loop such that the values of LO and UP are computed once before the repetitions of BODY, giving the values vlo and vup, respectively. vlo and vup determine the repetition pattern of the loop even in cases in which LO or UP is affected by the execution of BODY. The loop variable i does not occur in PRE, POST, LO and UP. During the execution of the loop i takes automatically the values in the interval [vlo, vup] in the appropriate order and cannot be written otherwise. Such controlled loops occur e.g. in Ada, Fortran, Modula and Pascal, and have recently also been introduced into C# and Java in the special form of the foreach-loop.

[2] This refers primarily to the generation of the VCs. The verification proper may still be rather difficult even for very simple statements: {True} skip; {Goldbach's conjecture}.

Basic idea and results.

In the SW engineering lifecycle the specification is the input for the implementation phase in which the program is created. Therefore, we make the assumption that the specification is correct[3] and that the newly created program may be incorrect. It would therefore be the best approach to infer a loop invariant from the specification only and then check whether the loop conforms to the specification. Our method comes near to this best approach by using only a small part of the loop and the specification (PRE, POST) for the inference of a loop invariant. This small part is the final bound of the FOR-loop, where the final bound of a FOR-loop is the upper bound for upwards counting loops and is the lower bound for downwards counting loops. The method works in two steps: (1) infer a hypothetical invariant HI and (2) try to prove the loop using HI as an invariant.

The use of the final bound is motivated by the observation that the last repetition of BODY usually establishes POST and that the value of the loop variable in this last repetition is the value of the final bound. Typically, this value is also the value of some constant c in POST, and this is then the reason that POST holds after this last repetition of BODY. Due to the approximating behavior of the FOR-loop it is a good heuristic to use $\mathrm{POST}^{c}_{loop\ variable}$ as a candidate for a loop invariant. This heuristic is called "*replacing a constant in the postcondition by a variable (RCPV)*" (Gries, 1983) and has been used by Dijkstra (1976) and Gries (1983) and by others for the manual development of WHILE-loops from specifications. Our method is a complete mechanization of RCPV for annotated FOR-loops. The range of applicability of the method is further extended by a technique called "bound transformation" and by taking common invariant conjuncts of PRE and POST into account. As a result, the method is applicable to the majority of FOR-loops occurring in practice.

In the examples we use a verification condition (VC) for the FOR-loops, which is an improvement of the proof rule of Hoare (1972) and has been first presented in Winkler (1998). There is no dependence between our method for the inference of HI and the VC used in the proof attempts.

Related work.

Soon after the seminal work on program verification by Floyd (1967) and Hoare (1969) began a phase of intensive work on developing methods for the determination of loop invariants, e.g. Wegbreit (1974), Caplain (1975), Katz and Manna (1976), Morris and Wegbreit (1977), Misra (1978), Basu (1980), Tamir (1980), Ellozy (1981), Gries (1982), Dunlop and Basili (1984), Mili et al. (1985), Paige (1986), Ernst et al. (1999), Kauer (1999), Ernst et al. (2000), Ball et al. (2001) and Flanagan and Qadeer (2002). More recently several methods have been presented to determine especially polynomial invariants (Müller-Olm and Seidl, 2003; Sankaranarayanan et al., 2004; Kovács and Jebelean, 2005; Seidl and Petter, 2005; Rodríguez-Carbonell and Kapur, 2006; Kovács, 2007). Some methods are for application by hand, e.g. Caplain (1975) and Misra (1978), some work in a semi-mechanized manner, e.g. Wegbreit (1974), Tamir (1980), Ball et al. (2001) and Flanagan and Qadeer (2002) and some are fully mechanized, e.g. Kauer (1999), Seidl and Petter (2005), Kovács and Jebelean (2005), Rodríguez-Carbonell and Kapur (2006) and Kovács (2007).

The different approaches exploit the annotated loop in different ways:

Some methods use the loop only, i.e. derive invariants from the code, e.g. Katz and Manna (1976), Basu (1980), Tamir (1980), Ellozy (1981), Paige (1986), Müller-Olm and Seidl (2003), Sankaranarayanan et al. (2004), Kovács and Jebelean (2005), Seidl and Petter (2005), Rodríguez-Carbonell and Kapur (2006), Kovács (2007) and Mili (2007). In Ernst et al. (1999) and Ernst et al. (2000) the loop is instrumented in order to output interesting variables ("trace variables"). The method then tries to infer an invariant from the values of the trace variables for several executions of the loop. This is a special case of deriving an invariant from the loop, because the values of the trace variables are determined by the loop. In Guo et al. (2006) BODY is executed symbolically a fixed number of times. Then recurrence analysis is applied to the results of these repetitions in order to try to synthesize a loop invariant. Symbolic execution and other techniques are used in Ireland (2007) and applied to pointer programs. After each symbolic execution of the loop body the method tries to find a fixed point.

[3] At least as long as not shown otherwise.

Another approach derives the invariant from the specification (Misra, 1978; Gries, 1982). Misra (1978) mentions two approaches: "A loop invariant could be a proposition about 'what has been done' or a proposition about 'what remains to be done'".

Gries (Gries, 1982, 1983) derives an invariant of the kind "what has been done" from POST. Gries attributes this methodology to Dijkstra (1976). Whereas Misra uses the invariant for the verification of an existing loop, Dijkstra and Gries use the invariant for the development of the loop itself. Janota (2007) derives invariants for WHILE-loops from the loop body and assertions contained therein. Wegbreit (1974) derives INV from POST and the loop condition of a WHILE-loop.

The method of Kauer (1999) is inspired by the use of RCPV in Gries (1982), where WHILE-loops are manually derived from the specification (PRE, POST) using RCPV. Kauer's method, on the other hand, is a fully mechanized version of RCPV and is tailored to the verification of an existing FOR-loop.

The rest of the paper is organized as follows. In Section 2 we present the verification condition for FOR-loops. Section 3 contains the method for the computation of an invariant and some examples of its application. Section 4 concludes the paper.

2. The verification condition for FOR-loops

The VC for FOR-loops in Winkler (1998) is based on the proof rule in Hoare (1972), which is depicted in (2):

$$\frac{a \le x \le b \ \&\ I([a \ldots x))\ \{Q\}\ I([a \ldots x])}{I([\,])\ \{\textbf{for}\ x := a\ \textbf{to}\ b\ \textbf{do}\ Q\}\ I([a \ldots b])} \tag{2}$$

I[s] is an assertion about the interval s and [ ] denotes the empty interval. This interval notation is used only in the discussion of the rule of Hoare (1972) in the first part of this section. In the rest of the paper [. . .] denotes universal quantification as used in Dijkstra and Scholten (1990).

The main differences are: (1) Winkler (1998) does not require I[ ] to hold before the first execution of the loop body. The invariant $I([LO \ldots i])$ need only hold after executions of the loop body. (2) the loop variable may occur in the invariant, and (3) the VC also works for loops with zero repetitions, i.e. $a > b$. Loop (7) is such a loop if $n = 0$. This strategy for the handling of the invariant is based on the following observations:

(1) the invariant is intended to be an assertion which is established by any execution of the loop body, especially the last one; therefore, it seems not necessary that the invariant holds before the FOR-loop. Collins (1988) calls such an invariant a "post-invariant";
(2) the invariant of a FOR-loop is typically an inductive assertion which involves the loop variable. In Hoare (1972) the loop variable must not occur in the invariant;
(3) in some programming languages the loop variable is declared locally in the loop and does not exist outside the loop, e.g. Algol 68, Ada, and C♯. If the invariant contains the loop variable and must hold before the loop, this could lead to illegal uses of the loop variable;
(4) there are examples in which it seems difficult to derive $I([\,])$ mechanically from $I([LO \ldots i])$. One example for this is in Winkler (1998, p. 8):

```
v := 5;
   -- v=5
FOR i in 1..10
LOOP v := i;                                   (3)
   -- inv ???
END LOOP;
   -- v=10
```

It is easy to see that $I([1 \ldots i]) \equiv v = i$ is an invariant which satisfies (3).

$I([1 \ldots i])^{i}_{10} \equiv I([1 \ldots 10]) \equiv v = 10$ is sufficient to establish the postcondition. We then have to determine $I([\,])$ such that

$$[\, v = 5 \Rightarrow I([\,])\,] \ \wedge\ [\, I([\,]) \Rightarrow wp(\text{“}v := 1;\text{”}, v = 1)\,] \tag{4}$$

holds. If we try $I([\,]) \equiv I([1 \ldots i])^i_{pred(1)} \equiv v = 0$ we observe that it does not work because $[v = 5 \Rightarrow v = 0] \equiv \mathit{False}$. On the other hand, $I([\,]) \equiv \mathit{true}$ does the trick; it is maximal in that it is the weakest solution of (4). But it is not derived mechanically from $I([1 \ldots i])$.

Apart from these differences, the VC (5), which is used in this paper, is expressed in a form suitable for automatic verification using the ABM, whereas the proof rule (2) in Hoare (1972) is formulated as a logical derivation rule and is not intended for use in automatic verification.

$$
\begin{aligned}
&[PRE \wedge LO > UP \Rightarrow POST] \ \wedge \\
&[PRE \wedge LO \leq UP \Rightarrow LO, UP \in Ti] \ \wedge \\
&[PRE \wedge LO \leq UP \Rightarrow wp(BODY^i_{LO}, INV^i_{LO})] \ \wedge \\
&[LO \leq i < UP \wedge INV \Rightarrow wp(BODY^i_{i+1}, INV^i_{i+1})] \ \wedge \\
&[LO \leq UP \wedge INV^i_{UP} \Rightarrow POST]
\end{aligned}
\tag{5}
$$

where

PRE	is the precondition
POST	is the postcondition
INV	is the invariant
Ti	is the value set of the type of the loop variable i
[. . .]	denotes universal quantification over the program variables and the specification variables

The VC (5) shows quite explicitly the different aspects of the verification of a FOR-loop:

first conjunct	: empty loop (i.e. zero repetitions of BODY)
second conjunct	: initialization of the loop
third conjunct	: first repetition of BODY
fourth conjunct	: further repetitions of BODY
fifth conjunct	: last repetition establishes POST

The VC (5) assumes that

(r1) the evaluation of LO and UP has no side effects

(r2) any evaluation of LO, UP or any of their subexpressions at any point in the FOR-loop yields the same value as in the initial evaluation at the beginning of the execution of the FOR-loop. This means especially that LO and UP are not written to in BODY and that they do not contain calls of functions which are not referentially transparent.

Both restrictions hold for many loops used in practice. Restriction (r2) is not severe; Winkler (1998) and Kauer and Winkler (2000) contain a VC which does not require restriction (r2) by introducing two fresh variables vlo and vup which are assigned the values of LO and UP before beginning the repetitions of the loop body. Since the method for the computation of the invariant does not depend on the exact form of the VC, we use the simpler form of the VC for the examples in this paper.

In Kauer and Winkler (2000) we show that (5) implies the correctness of the loop (1) and that the correctness of (1) implies the existence of an invariant INV satisfying (5).

3. A method for computing invariants of FOR-loops

3.1. Basic idea

The general wp-rule for a FOR-loop cannot always be solved exactly. Usually, some weaker form of correctness is used which uses a loop invariant (Gries, 1983; Winkler, 1998). This means that the

engineer has to determine a suitable invariant. If such an invariant can be computed mechanically the task of the engineer will be easier. In this section we present a method for the mechanical inference of invariants of FOR-loops, which are annotated with PRE and POST only.

The method is an extension and mechanization of the heuristic RCPV (Dijkstra, 1976; Gries, 1983), where in our case the variable is always the loop variable. The heuristic RCPV is typically applied by replacing the final bound in POST by the loop variable. This is due to the observation that the final bound is also the final value of the loop variable in the last repetition of BODY, if the loop does not terminate prematurely. If the final bound occurs in POST then the last value of the loop variable plays the same role in POST. For many FOR-loops the proposition made in POST about the value of the final bound holds for the value of the loop variable in the intermediate stages of the execution of the loop. This observation leads to the strategy RCPV where the final bound plays the role of the constant. This basic method is extended to many practically relevant cases in which the final bound is not a constant but a somewhat more complicated expression, which need not occur in POST (see Section 3.2).

For upwards counting loops the final bound is UP, and for downwards counting loops it is LO. In the following we show the derivation of the method for upwards counting loops. The method can be adapted to downwards counting loops quite easily. For the old method of Kauer and Winkler (2007) this is shown in Kauer and Winkler (2000). The algorithm in Section 3.5 works for both kinds of loops.

The method works in two steps:

(1) try to infer a predicate HI (hypothetical invariant) from the AFL. There are cases in which the method does not generate a predicate HI, e.g. if UP contains function calls. A check of Gonnet and Baeza-Yates (1991) gave the following result: in most FOR-loops UP has one of the forms: (a) variable, (b) sum of two variables, or (c) sum of a variable and a constant. This book is mostly on non-numerical algorithms which often use WHILE-loops. There are 166 FOR-loops and 116 of them are appropriate for our method. A similar check of Engeln-Müllges and Reutter (1993), a book on numerical algorithms in Fortran-77, gives an even better result: there are 958 DO-loops and 947 (=98.85%) of these are suitable for our method. In 707 DO-loops the final bound has one of the two forms: literal (122) or variable (585).

(2) try to prove $(\text{FOR-VC})_{HI}^{INV}$. There are three possible answers:
   (a) the proof succeeds, i.e. HI is an invariant and the loop is correct.
   (b) the refutation succeeds. In this case HI may or may not be an invariant.
   (c) neither proof nor refutation succeed, i.e. the prover "gives up" or does not terminate. In this case it is unknown whether the loop is correct or incorrect, or whether HI is or is not an invariant.

   Only in case (a) does the method say that the loop is correct.

The idea behind this method is that most FOR-loops compute their final result by computing a sequence of partial intermediate results which approximate the final result better and better. The final result is described by POST and often depends on a characteristic term which usually is UP. POST_i^{UP}, which then depends on i, often characterizes these partial results. Substitution is usually only defined for elementary terms i.e. variables and literals. By solving UP for such an elementary term RCPV can also be applied if UP is a more complicated term. This is called bound transformation and is the topic of Section 3.2.

A very simple example to demonstrate the method is a loop for the summation of the first 100 natural numbers:

```
   -- PRE: s = 0 ∧ s ∈ int32
FOR i in 1..100 LOOP s := s+i; end loop;                    (6)
   -- POST: s = ⟨Σj: 1..100: j⟩ ∧ s ∈ int32
```

In (6) we assume that the type of s is int32. In (6) RCPV can be applied directly and gives the HI

$$HI \equiv POST^{UP}_{loop\ variable} \equiv POST_i^{100} \equiv \langle s = \langle \Sigma j : 1..100 : j\rangle \wedge s \in int32\rangle_i^{100}$$
$$\equiv s = \langle \Sigma j : 1..i : j\rangle \wedge s \in int32$$

The loop (6) with the invariant HI satisfies (5) and therefore, HI is an invariant of (6) and (6) is correct. Since $\langle \forall i \in 0..100 : s = \langle \Sigma j : 1..i : j \rangle \wedge s \in int32 \rangle$ holds, we could have omitted `s ∈ int32` in (6). We included it for documentation purposes.

(6) is a very special loop because the upper bound is the fixed number 100. Often the upper bound will be a program variable whose value is constant in the FOR-loop. Such a more general FOR-loop is given in (7).

```
  -- PRE: s=0 ∧ 0≤n≤65535 ∧ n=N
FOR i in 1..n LOOP s := s+i; end loop;                                    (7)
  -- POST: s = ⟨Σj: 1..n: j⟩ ∧ s ∈ int32 ∧ 0≤n≤65535 ∧ n=N
```

We assume that s and n are of type int32. N is a specification variable which is used to guarantee that the value of n after the loop is the same as that before the loop. If we compute HI mechanically as POST_i^n we obtain

$$\begin{aligned} HI &\equiv POST_i^n \equiv \langle s = \langle \Sigma j : 1..n : j \rangle \wedge s \in int32 \wedge 0 \le n \le 65535 \wedge n = N \rangle_i^n \\ &\equiv s = \langle \Sigma j : 1..i : j \rangle \wedge s \in int32 \wedge 0 \le i \le 65535 \wedge i = N \end{aligned}$$

We observe immediately that HI is not an invariant of (7) because i has not always the value N (if $N > 1$).

A strategy for avoiding this problem is to apply the substitution $(n \mapsto i)$ only to those conjuncts of POST which are not also a conjunct of PRE. In the example this results in

$$\begin{aligned} HI &\equiv \langle s = \langle \Sigma j : 1..n : j \rangle \wedge s \in int32 \rangle_i^n \wedge 0 \le n \le 65535 \wedge n = N \\ &\equiv s = \langle \Sigma j : 1..i : j \rangle \wedge s \in int32 \wedge 0 \le n \le 65535 \wedge n = N \end{aligned}$$

The loop (7) with HI as invariant satisfies (5).

A method for the identification of common conjuncts is given in Section 3.3.

3.2. Bound transformation

The method $\text{POST}^{UP}_{loop\ variable}$ works only if UP is a constant or a variable. This is a considerable restriction. One idea is to insert the assignment "`vup := UP;`" immediately before the loop, where vup is a fresh variable, and then use vup as the upper bound. But this does not work in general, since vup does not occur in POST. Substituting a non-occurring variable does not change POST, so that POST itself had to be considered as an invariant, which does not work in most cases.

In the loop (8) UP is not a variable but a more complicated expression.

```
  -- PRE: s=0 ∧ 0≤m ∧ 0≤n ∧ m+n≤65535
FOR i in m..m+n LOOP s := s+i; end loop;                                  (8)
  -- POST: s = ⟨Σj: m..m+n: j⟩ ∧ s ∈ int32 ∧ 0≤m ∧ 0≤n ∧ m+n≤65535
```

The restriction $m + n \le 65535$ is somewhat too sharp for the requirement $s \in int32$ in POST. If $n = 0$ then $m \in [0, int32'Last]$ implies $s \in int32$. This over-restriction of m+n has no influence on the mechanism of bound transformation in this example.

The introduction of vup and the application of RCPV results in

```
  -- s=0 ∧ 0≤m ∧ 0≤n ∧ m+n≤65535
vup := m+n;
  -- PRE: s=0 ∧ 0≤m ∧ 0≤n ∧ m+n≤65535 ∧ vup=m+n                            (9)
  --    ≡ sp(s=0 ∧ 0≤m ∧ 0≤n ∧ m+n≤65535, ‘‘vup:=m+n;’’)
FOR i in m..vup LOOP s := s+i; end loop;
  -- POST: s = ⟨Σj: m..m+n: j⟩ ∧ s ∈ int32 ∧ 0≤m ∧ 0≤n ∧ m+n≤65535
```

Since vup does not occur in POST and $C_{com} \equiv 0 \le m \wedge 0 \le n \wedge m + n \le 65535$ are the common conjuncts of PRE and POST the hypothetical invariant is

$$\begin{aligned} HI &\equiv POST'^{n}_{i} \wedge C_{com} \\ &\equiv \langle s = \langle \Sigma j : m..m+n : j\rangle \wedge s \in int32\rangle^{vup}_{i} \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535 \\ &\equiv s = \langle \Sigma j : m..m+n : j\rangle \wedge s \in int32 \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535 \\ &\equiv POST \end{aligned}$$

where POST' is POST without the common conjuncts C_{com}. It is easy to see that HI is not an invariant of (9).

Another possibility is to find a transformation $t \in E \to E$ for which $r(t(UP))$ is a simple variable or constant occurring free in UP and POST, where E is the set of expressions and $r \in E \to E$ is a function which reduces (simplifies) its argument to an irreducible expression. This variable or constant is called the pivot element.

For UP = "m+n" such a transformation is $t(e)$ = "e−m". We obtain t("$m+n$") = "$m+n-m$" and $r(t$("m+n")) = r("m+n−m") = "n". The pivot element n occurs free in UP and in POST. The idea is now to substitute n by some expression involving the loop variable i in order to obtain a hypothetical invariant. Since the value of UP is the value of i in the last repetition of BODY, the value of $t(UP)$ is the value of t("i") = "i−m" in this last repetition. This gives the following hypothetical invariant for (8):

$$\begin{aligned} HI &\equiv POST'^{r(t(UP))}_{r(t(i))} \wedge C_{com} \equiv POST'^{n}_{i-m} \wedge C_{com} \qquad (10) \\ &\equiv \langle s = \langle \Sigma j : m..m+n : j\rangle \wedge s \in int32\rangle^{n}_{i-m} \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535 \\ &\equiv s = \langle \Sigma j : m..m+i-m : j\rangle \wedge s \in int32 \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535 \\ &\equiv s = \langle \Sigma j : m..i : j\rangle \wedge s \in int32 \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535 \end{aligned}$$

HI and (8) together satisfy (5), which shows that the loop in (8) is correct and HI is an invariant of this loop.

The net effect of $POST'^{n}_{i-m}$ in (10) is the substitution of "m+n" by "i". In this special case this could also have been obtained by

$$HI \equiv POST'^{UP}_{i} \wedge C_{com}$$

because the term UP occurs as a subterm in POST'. But this direct form of substitution is not applicable if UP does not occur explicitly as a subterm in POST'. Examples for this are the postconditions:

$$POST_2 \equiv s = \langle \Sigma j : m..n+m : j\rangle \wedge s \in int32 \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535$$

$$POST_3 \equiv s = \langle \Sigma j : 0..n : j+m\rangle \wedge s \in int32 \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535$$

which are logically equivalent to POST.

The transformation $t(e) =$ "$e-m$" works also for $POST_2$ and $POST_3$.

There is a second transformation which transforms UP of (8) into a variable: $t2(e) =$ "$e-n$". The application of $t2$ gives

$$HI \equiv s = \langle \Sigma j : i-n..i : j\rangle \wedge s \in int32 \wedge 0 \le m \wedge 0 \le n \wedge m+n \le 65535$$

but this is not an invariant of (8).

The general strategy of bound transformation is to try to find a transformation $t \in E \to E$ such that $r(t(UP)) =$ "v", $r(t(UP)) =$ "−v" or $r(t(UP)) =$ "1/v" for some $v \in free(UP) \cap free(POST)$.

Since $r(t(UP))$ has this simple form we get the hypothetical invariant

$$\begin{aligned} HI &\equiv POST'^{v}_{r(t(i))} \wedge C_{com} \qquad \text{if } r(t(UP)) = \text{"v"}\text{, or} \\ HI &\equiv POST'^{v}_{-r(t(i))} \wedge C_{com} \qquad \text{if } r(t(UP)) = \text{"}-\text{v"}\text{, or} \qquad (11) \\ HI &\equiv POST'^{v}_{1/r(t(i))} \wedge C_{com} \qquad \text{if } r(t(UP)) = \text{"1/v"} \end{aligned}$$

where C_{com} are the common conjuncts of PRE and POST. Common conjuncts will be treated in Section 3.3.

The transformation t depends on UP and v and is a value of $T \in E \times Var \to (E \to E)$, i.e. $t = T(UP, v) \in E \to E$. The determination of $T(UP, v)$ is similar to solving an equation for one of the terms "v", "−v" or "1/v". In order to find t for a given expression UP and a given pivot element v we determine the syntactic transformation necessary to semantically neutralize all terms apart from "v", "−v" or "1/v", respectively. For example if UP = "m+10" we obtain T(UP, m)(e) = "e−10" and for UP = "n∗a+b" we obtain T(UP, n)(e) = "(e−b)/a" and T(UP, b)(e) = "e−n∗a". For $UP =$"$NX + M * (M + 1)/2$", which is one of the most complicated upper bounds in Engeln-Müllges and Reutter (1993), we get $T(UP, M)(e) =$ "$\sqrt{(e - NX) * 2 + 1/4} - 1/2$".

Table 1 lists the transformations which are sufficient for 941 DO-loops of the 958 DO-loops in Engeln-Müllges and Reutter (1993).

Table 1
Transformations $t = T(UP, v)$ for different forms of UP.

Form of UP	o_1 v	e_1 o_2 v	o_1 v o_2 e_1	e_1 o_2 v o_3 e_2
$T(UP, v)(e)$	e	$e - e_1$	$e\ o_2^{-1}\ e_1$	$e - e_1\ o_3^{-1}\ e_2$
$r(T(UP, v)(UP))$	o_1 v	o_2 v	o_1 v	o_2 v

Form of UP	v o_4 e_1	v o_4 e_1 + e_2	e_1 + v o_4 e_2
$T(UP, v)(e)$	$e\ o_4^{-1}\ e_1$	$(e - e_2)\ o_4^{-1}\ e_1$	$(e - e_1)\ o_4^{-1}\ e_2$
$r(T(UP, v)(UP))$	v	v	v

Form of UP	e_1 ∗ v	e_1 / v	e_1 ∗ v + e_2	e_1 / v + e_2
$T(UP, v)(e)$	e / e_1	e / e_1	$(e - e_2)$ / e_1	$(e - e_2)$ / e_1
$r(T(UP, v)(UP))$	v	1/v	v	1/v

where v $\notin$ free(e_1) $\cup$ free(e_2), $o_1 \in \{+, -, \epsilon\}$, $o_2, o_3 \in \{+, -\}$, $+^{-1} = -$, $-^{-1} = +$, $o_4 \in \{*, /\}$, $*^{-1} = /$, $/^{-1} = *$ and e_1 and e_2 are parenthesized expressions. If e.g. UP = "v−a+b" we assume that UP has been transformed into "v−(a−b)" or an equivalent parenthesized form.

3.3. Determination of common conjuncts

According to the observation in (7) we present a refinement of the basic strategy by exempting common invariant conjuncts from RCPV. Such conjuncts often occur in programs with nested loops. One example is example 17 in Freining et al. (2002). A second example is the algorithm (12), which computes the ∞-norm p of the matrix a of size $m \times n$, which is defined in Golub and Loan (1990) as:

$$p = \langle Max : 1 \le k \le m : \langle \Sigma\, c : 1 \le c \le n : |a(k, c)| \rangle \rangle$$

```
   -- PREo: m, n ≥ 1 ∧ p = 0
FOR i IN 1..m LOOP
   s := 0;
      -- PREi: s=0 ∧ p = ⟨Max k: 1..i-1: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
   FOR j IN 1..n LOOP
      s := s + abs(a(i,j));
   END LOOP;                                                    (12)
      -- POSTi: s = ⟨Σ c: 1..n: |a(i,c)|⟩ ∧
      --         p = ⟨Max k: 1..i-1: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
   IF s > p THEN p := s; END IF;
      -- p = ⟨Max k: 1..i: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
END LOOP;
   -- POSTo: p = ⟨Max k: 1..m: ⟨Σ c: 1..n: |a(k,c)|⟩⟩
```

PREi and POSTi have one conjunct in common: C ≡ p = ⟨Max k: 1..i: ⟨Σ c: 1..n: |a(k,c)|⟩⟩. It is easy to see that in C the upper bound n of the inner loop must not be replaced by j to obtain an HI. We obtain therefore the HI ≡ s = ⟨Σ c: 1..i: |a(i,c)|⟩ ∧ C which is an invariant of the inner loop.

C is a syntactically common conjunct (SCC) of PREi and POSTi. Additionally, C is also semantically invariant because all free variables of C are never changed in the inner loop. In fact, all these variables are only read in the inner loop, but never written. Such SCC, which are also semantically invariant, are used as common conjuncts (CC) in our method. A simple condition for semantic invariance of an SCC C is: noWrite(BODY, free(C)) which means that no free variable of C is ever written in BODY. This condition is typically fulfilled in practical cases.

We determine common conjuncts as follows

(a) transform PRE and POST into a normal form. The exact definition of this normal form is not important for our method. An example of such a normal form has been given in Kauer and Winkler (2000).
(b) determine the syntactically common conjuncts $C_1, \ldots, C_n$.
(c) determine those C_i for which noWrite(BODY, free(C_i)) holds.
(d) let C_{com} be the conjunction of these C_i.

If the method finds any invariant common conjuncts the normalized POST can be written as POST' ∧ C_{com} , where POST' does not contain any conjunct of C_{com}. The hypothetical invariant is then that given above in (11).

3.4. Specialization of the verification condition

Because HI depends on the given AFL a VC for this AFL may be simpler than the general VC in (5). If HI ≡ $POST'^{v}_{r(t(i))} \wedge C_{com}$ then it is obvious that the fifth conjunct in (5) is always True:

$$
\begin{aligned}
& [LO \le UP \wedge INV^{i}_{UP} \Rightarrow POST] \\
& \text{-- } INV \equiv HI \equiv POST'^{v}_{r(t(i))} \wedge C_{com} \\
\equiv\ & [LO \le UP \wedge \langle POST'^{v}_{r(t(i))} \wedge C_{com} \rangle^{i}_{UP} \Rightarrow POST] \\
\equiv\ & [LO \le UP \wedge \langle POST'^{v}_{r(t(i))} \rangle^{i}_{UP} \wedge C_{com}{}^{i}_{UP} \Rightarrow POST] \\
& \text{-- } i \notin free(POST) \Rightarrow i \notin free(POST') \wedge i \notin free(C_{com}) \\
\equiv\ & [LO \le UP \wedge POST'^{v}_{r(t(UP))} \wedge C_{com} \Rightarrow POST] \\
& \text{-- } r(t(UP)) = \text{“}v\text{”} \\
\equiv\ & [LO \le UP \wedge POST'^{v}_{v} \wedge C_{com} \Rightarrow POST] \\
\equiv\ & [LO \le UP \wedge POST' \wedge C_{com} \Rightarrow POST] \\
& \text{-- } POST' \wedge C_{com} \equiv POST \\
\equiv\ & [LO \le UP \wedge POST \Rightarrow POST] \\
\equiv\ & True
\end{aligned}
$$

Such simplifications will reduce the effort for the correctness proof and thus speed up the APV. The substitution of INV by HI in (5) for the case r(t(UP)) = “v” gives the specialized VC (13):

$$
\begin{aligned}
& [PRE \wedge LO > UP \Rightarrow POST'] \wedge \\
& [PRE \wedge LO \le UP \Rightarrow LO, UP \in Ti] \wedge \\
& [PRE \wedge LO \le UP \Rightarrow wp(BODY^{i}_{LO}, POST'^{r(t(UP))}_{r(t(LO))} \wedge C_{com})] \wedge \\
& [LO \le i < UP \wedge POST'^{r(t(UP))}_{r(t(i))} \wedge C_{com} \Rightarrow wp(BODY^{i}_{i+1}, POST'^{r(t(UP))}_{r(t(i+1))} \wedge C_{com})]
\end{aligned}
\qquad (13)
$$

3.5. Algorithm for the application of the method

We are now ready to put the pieces together and present the application of the method as an algorithm, which works for both upwards and downwards counting AFLs.

```
   -- input: PRE, POST, i, LO, UP, BODY, UpwardsCounting?
AFLProved: enum(proof, noproof) := noproof;
FinalBound: expression;
if UpwardsCounting? then FinalBound := UP; else FinalBound := LO; end if
if FinalBound is suitable (see Table 1)
then C_com: expression := true;
       POST': expression := POST;
       if there is an SCC c with noWrite(BODY, free(c))
       then C_com := ⟨⋀ c: c is an SCC: noWrite(BODY, free(c))⟩;
            POST' := con(set(POST) – set(C_com));
       end if
          -- create the set T of all possible transformations t(FinalBound, v),
          -- where v ∈ free(FinalBound) ∩ free(POST)
       for each t ∈ T do      -- r(t(FinalBound)) ∈ {v, −v, 1/v}
          case r(t(FinalBound)) in
             when v     => POST' := POST'^v_{r(t(i))};
             when −v    => POST' := POST'^v_{−r(t(i))};
             when 1/v   => POST' := POST'^v_{1/r(t(i))};
          end case
          if the AFL can be proved using POST' and C_com
             using the appropriate VC (Section 3.4)
          then AFLProved := proof; exit;
          end if
       end for
end if
   -- output: AFLProved
```

The functions con(·) and set(·) are defined as follows:

$\mathrm{set}(C_1 \wedge \cdots \wedge C_n) = \{C_1, \ldots, C_n\}$,

$\mathrm{con}(\{C_1, \ldots, C_n\}) = C_1 \wedge \cdots \wedge C_n$

3.6. Examples

In the following example (14) the natural numbers in the range 1 .. 2∗m+1 are summed up.

```
   -- PRE: s=0 ∧ 0≤m≤32767
FOR i in 1..2∗m+1 LOOP
   s := s+i;                                                    (14)
end loop;
   -- POST: s = ⟨Σj: 1..2∗m+1: j⟩ ∧ s ∈ int32 ∧ 0≤m≤32767
```

UP is suitable, $C_{com} \equiv 0 \le m \le 32767$, free(UP) ∩ free(POST) = {m}, there is one transformation:

T("2∗m+1", m) = "(e−1)/2"

which gives the HI:

$s = \langle \Sigma j: 0..i: j \rangle \wedge s \in \mathrm{int32} \wedge 0 \le m \le 32767$

which is an invariant of (14). (14) and HI satisfy (13).

The second example is from Seidl and Petter (2005) and computes the sum of squares of the first n natural numbers. An equivalent AFL is (15).

$$
\begin{array}{l}
\texttt{-- PRE: } 0 \le n \le 1860 \ \wedge\ n = N \ \wedge\ x = 0 \\
\texttt{FOR y in 0..n LOOP} \\
\quad \texttt{x := y*y + x;} \\
\texttt{end loop;} \\
\quad \texttt{-- POST: } x = (2n^3+3n^2+n)/6 \ \wedge\ x \in \texttt{int32} \ \wedge\ 0 \le n \le 1860 \ \wedge\ n = N
\end{array}
\tag{15}
$$

Additionally to Seidl and Petter (2005) we assume that $x \in \text{int32}$ and use n in POST instead of y, which may not be in scope. Since UP is a simple variable we obtain directly

$$\text{HI} \ \equiv\ x = (2y^3+3y^2+y)/6 \wedge x \in \text{int32} \wedge 0 \le n \le 1860 \ \wedge n = N$$

HI is an invariant of (15), and the loop (15) together with HI satisfies (13).

4. Conclusion

We have developed a method for the mechanical inference of invariants for a practically relevant class of FOR-loops. The method can be incorporated into automatic program verifiers and would lead to a simplification of program verification using such a tool. By extending the suitable forms of the final bound the applicability of the method could be extended to further classes of FOR-loops.

Acknowledgements

The authors would like to thank the anonymous referees for a number of very helpful hints and remarks.

References

Ahrendt, W., Baar, T., Beckert, B., et al., 2005. The key tool. Software Syst. Model 4, 32–54. doi:10.1007/s10270-004-0058-x.
Back, R.-J., 2006. Invariant based programming. In: LNCS, vol. 4024. Springer, Berlin, pp. 1–18. doi:10.1007/11767589_1.
Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K., 2001. Automatic predicate abstraction of C programs. In: ACM PLDI '01, pp. 203–213. doi:10.1145/378795.378846.
Barnes, J., 2000. High Integrity Ada – The SPARK Approach –. Addison-Wesley, Harlow, etc., ISBN: 0-201-17517-7.
Barnett, M., Chang, B.-Y.E., DeLine, R., et al., 2006. Boogie: A modular reusable verifier for object-oriented programs. In: LNCS, vol. 4111. Springer, Berlin, pp. 364–387. doi:10.1007/11804192_17.
Basu, S.K., 1980. A note on synthesis of inductive assertions. IEEE TSE 6 (1), 32–39. doi:10.1109/TSE.1980.230460.
Caplain, M., 1975. Finding invariant assertions for proving programs. SIGPLAN Not. 10 (6), 165–171. doi:10.1145/800027.808436.
Collins, W.J., 1988. The trouble with FOR-loop invariants. ACM SIGCSE Bull. 20 (1), 1–4. doi:10.1145/52965.52966.
Dijkstra, E.W., 1976. A Discipline of Programming. Prentice-Hall Inc, Englewood Cliffs, ISBN: 0-13215871-X.
Dijkstra, E.W., Scholten, C.S., 1990. Predicate Calculus and Program Semantics. Springer, New York, etc., ISBN: 0-387-96957-8.
Dunlop, D.D., Basili, V.R., 1984. A heuristic for deriving loop functions. IEEE TSE 10 (3), 275–285.
Ellozy, H.A., 1981. The determination of loop invariants for programs with arrays. IEEE TSE 7 (2), 197–206. doi:10.1109/TSE.1981.234517.
Engeln-Müllges, G., Reutter, F., 1993. Numerik-Algorithmen mit FORTRAN 77-Programmen. (Numerical Algorithms with Fortran 77 Programs). In: Aufl., vol. 7. BI Wissenschaftsverlag, Mannheim usw, ISBN: 3-411-15117-X, English edition 1996, ISBN 3-540-60529-0.
Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 1999. Dynamically discovering likely program invariants to support program evolution. In: ICSE '99, pp. 213–224. doi:10.1145/302405.302467.
Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D., 2000. Quickly detecting relevant program invariants. In: ICSE 2000. pp. 449–458, doi:10.1145/337180.337240.
Flanagan, C., Qadeer, S., 2002. Predicate abstraction for software verification. In: ACM POPL'02. pp. 191–202. doi:10.1145/503272.503291.
Floyd, R.W., 1967. Assigning meaning to programs. In: Schwartz, J.T. (Ed.), Mathematical Aspects of Computer Science. AMS, ISBN: 0-8218-1319-6, pp. 19–32.
Freining, C., Kauer, S, Winkler, J.F.H., 2002. Ein Vergleich der Programmbeweiser FPP, NPPV und SPARK. (A Comparison of the Program Provers FPP, NPPV and SPARK). In: Ada-Deutschland Tagung 2002. Shaker Verlag, Aachen, ISBN: 3-8265-9956-X, pp. 127–145.
Giese, M., Jebelean, T. (Eds.), 2007. WING 2007 - Workshop on Invariant Generation. RISC-Linz Report 07-07, Linz, Austria.
Golub, G.H., Loan, C.F.van, 1990. Matrix Computations, Second ed. John Hopkins Univ. Press, Baltimore, etc., ISBN: 0-8018-3739-1, second pr.

Gonnet, G.H., Baeza-Yates, R., 1991. Handbook of Algorithms and Data Structures. Addison Wesley, Wokingham, ISBN: 0-201-41607-7.
Gries, D., 1982. A Note on a Standard Strategy for Developing Loop Invariants and Loops. Sci. Comput. Program. 2, 207–214. doi:10.1016/0167-6423(83)90015-1.
Gries, D., 1983. The Science of Programming. Springer, New York, ISBN: 0-387-90641-X, Second pr.
Gumm, H, 1999. Generating algebraic laws from imperative programs. Theoret. Comput. Sci. 217, 385–405. doi:10.1016/S0304-3975(98)00278-3.
Guo, B., Vachharajani, N., August, D.I., 2007. Shape analysis with inductive recursion synthesis. In: ACM PLDI '07, pp. 256–265. doi:10.1145/1250734.1250764.
Hoare, C.A.R., 1969. An axiomatic basis of computer programming. CACM 12 (10), 576–580. doi:10.1145/363235.363259. 583.
Hoare, C.A.R., 1972. A note on the FOR statement. BIT 12, 334–341. doi:10.1007/BF01932305.
Ireland, A., 2007. A cooperative approach to loop invariant discovery for pointer programs. In Giese and Jebelan (2007) pp. 2–14.
Janota, M., 2007. Assertion-based loop invariant generation. In Giese and Jebelan (2007) pp. 15–26.
Katz, S., Manna, Z., 1976. Logical analysis of programs. CACM 19 (4), 188–206. doi:10.1145/360032.360048.
Kauer, S., 1999. Automatische Erzeugung von Verifikations- und Falsifikationsbedingungen sequentieller Programme. (Automatic Generation of Verification Conditions and Falsification Conditions of Sequential Programs). Dissertation, Friedrich Schiller University, Jena, Germany.
Kauer, S., Winkler, J.F.H., 1999. FPP: An automatic program prover for ada statements. In: Workshop Objektorientierung und sichere Software mit Ada, Karlsruhe, Germany, 21–22 Apr. 1999.
Kauer, S., Winkler, J.F.H., 2000. Automatic generation of invariants for FOR-loops based on an improved proof rule. Report Math/Inf/2000/26. Friedrich Schiller University, Dept. of Math. & Comp. Sci., Jena, Germany.
Kauer, S., Winkler, J.F.H., 2007. Mechanical generation of invariants for FOR-loops. In Giese and Jebelan (2007) pp. 27–40.
Kovács, L., 2007. Automated polynomial invariant generation by algebraic techniques for imperative program verification in theorema. In Giese and Jebelan (2007) pp. 56–69.
Kovács, L.I., Popov, N., Jebelean, T., 2003. Verification of imperative programs in theorema. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria.
Kovács, L.I., Jebelean, T., 2005. An algorithm for automated generation of invariants for loops with conditionals. Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria.
Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., 2008. JML Reference Manual. Draft, Rev. 1.231, 2008.05.13,, visited 2008 Jun 27. www.eecs.ucf.edu/~leavens/JML/OldReleases/jmlrefman.pdf.
Mili, A., 2007. Reflexive transitive loop invariants: A basis for computing loop functions. In Giese and Jebelan (2007) pp. 100–114.
Mili, A., Desharnais, J., Gagné, J.-R., 1985. Strongest invariant functions: Their use in the systematic analysis of while-statements. Acta Inform. 22 (1), 47–66. doi:10.1007/BF00290145.
Misra, J., 1978. Some aspects of the verification of loop computations. IEEE TSE 4 (6), 478–486. doi:10.1109/TSE.1978.233871.
Morris Jr., J.H., Wegbreit, B., 1977. Subgoal induction. CACM 20 (4), 209–222. doi:10.1145/359461.359466.
Müller-Olm, M., Seidl, H., 2003. Computing polynomial program invariants. October 2, 2003, visited 2007 Feb 11. www.informatik.fernuni-hagen.de/forschung/informatikberichte/pdf-versionen/310.pdf.
Paige, R., 1986. Programming with invariants. IEEE Software 3 (1), 56–69. doi:10.1109/MS.1986.233070.
Rodríguez-Carbonell, E., Kapur, D., 2006. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. 64 (1), 54–75. doi:10.1016/j.scico.2006.03.003. Available online 28 Sept 2006 at www.sciencedirect.com/, visited 2007. Jan. 22.
Sankaranarayanan, S., Sipma, H.B., Manna, Z., 2004. Non-linear loop invariant generation using Gröbner bases. In: ACM POPL '04. pp. 318–329. doi:10.1145/964001.964028.
Seidl, H., Petter, M., 2005. Inferring polynomial invariants with polyinvar. Technische Universität München, Garching, Germany, visited 2007. Feb. 12. www2.cs.tum.edu/~petter/papers/nsad05.pdf.
Tamir, M., 1980. ADI: Automatic derivation of invariants. IEEE TSE 6 (1), 40–48. doi:10.1109/TSE.1980.230461.
Turing, A., 1949. Checking a large routine. In: Williams, L.R., Campbell-Kelly, M. (Eds.), The Early British Computer Conferences. MIT Press, Cambridge, ISBN: 0-262-23136-0, pp. 70–72. 1989.
Wegbreit, B., 1974. The synthesis of loop predicates. CACM 17 (2), 102–112. doi:10.1145/360827.360850.
Winkler, J.F.H., 1997. The Frege Program Prover. 42. In: Int. Wiss. Koll., Ilmenau., (ISSN: 0943-7207), vol. 1. pp. 116–121.
Winkler, J.F.H., 1998. New proof rules for FOR-loops. Report Math/Inf/98/13. Friedrich Schiller University, Dept. of Math. & Comp. Sci., Jena, Germany.