Generating Loop Invariants for Imperative Programs

Laura Ildikó Kovács
Research Institute for Symbolic Computation, Austria
kovacs@risc.upb.de

- Program Verification
- The Theorema System
- Imperative Program Verification in Theorema
- Generation of Invariants
- Generation of Termination Terms
- Conclusion and Future Work
Program Verification

Relationship: Imperative | Rule-based Programs

Rule-based Programs

- specifications, programs and verification can be viewed within a single uniform framework.
- (consequence) verification: checking that each clause is true in the intended model of the program.

Imperative Programs

- additional assertions are needed, such as for ex. loop invariants
- forward reasoning using the inductive assertion method [Hoare69, Floyd77]
  Forward reasoning is typified by the construction of iterative algorithms - the initial condition:
  Predicate Transformer method (using weakest precondition) [Dijkstra76], [Gries81]: equivalent proof of partial correctness, but also used for proving termination and for deriving
  - backward reasoning using the subgoal induction [MorrisWeg77]
  Backward reasoning is typified by the construction of recursive algorithms
  - the base cases are defined
  - more complex initial conditions are handle recursively solving simpler problems and applying
  Less-known verification method for imperative programs: subgoal induction:
  basic idea:- rather than using the value of a variable had in the "past", we use the value that it
  Subgoal induction:
  Proves - invariant holds at the end of loop
  - if invariant holds at iteration N+1, then it must have held at iteration N.

If invariants are given, [ClarkEmden81]
  subgoal induction (iterative imperative programs) ≡ top-down evaluation (rule-based programs)
  inductive assertion (iterative imperative programs) ≡ bottom-up evaluation (rule-based programs)

Example: GCD

Imperative Alg.
WHILE [x ≠ y, 
    IF [x > y
        x := x - y,
        y := y - x
    ]
]
answer := x

Proof of correctness [Naish90, Hoare69]:
- introduce "shadow variables": \( x_0, y_0 \) (initial values \( x, y \))
- loop invariant is formulated: \( \gcd(x, y) = \gcd(x_0, y_0) \)
- inductive verification: \( \{ \text{if the invariant holds (at the end) then the answer is correct} \) 
  \( \text{the invariant hold initially and after each loop iteration} \)

- predicate transformer (wp - [Futshek89, Hoare69, AptO97]): termination is only: \( x = y \)
  we know: \( x = y \land \gcd(x, y) = \gcd(x_0, y_0) \Rightarrow x = \gcd(x_0, y_0) \)
  proof that invariant holds is a little more complex
  - working backwards, to show that holds at the end of loop,
    suffices to show at the start of the loop:
    \[
    \bigvee \left\{ (x > y) \land \gcd(x_0, y_0) = \gcd(x - y, y) \right\} \\
    \left\{ (x \leq y) \land \gcd(x_0, y_0) = \gcd(x, y - x) \right\}
    \]
    and this condition is implied by:
    \( \gcd(x, y) = \gcd(x_0, y_0) \land x \neq y \)

Rule-based Alg. (the logic behind the imperative alg. - i.e. invariant- is exposed)
(while loop is converted in a recursive procedure)

\[
\text{GcD}[\_\_, \_\_] := X \quad (*\text{exit cond}* )
\]

\[
\text{GcD}[\_\_, \_\_] /; X > Y := \text{GcD}[X - Y, Y] \quad (*\text{if-then}* )
\]

\[
\text{GcD}[\_\_, \_\_] := \text{GcD}[X, Y - X] \quad (*\text{else}* )
\]

Proof of partial correctness [Naish90]:
- show that each clause is a true statement about the gcd, i.e. each clause is a consequence of its
  ex: (*if-then*) correct \( (x > y) \land GcD(x - y, y) = z \Rightarrow GcD(x, y) = z \)
- imperative alg. uses "shadow vars" \( x_0, y_0 \). Logic alg. never refers to the top-level (initial) va

Program Verification Systems
EHDM, PVS, STeP, Sunrise, Daikon, SPARK, SPIN, Theorema, ...
The **Theorema System**

*Theorema* - a computer aided mathematical assistant for automated reasoning

- Computing (simplifying)
- Solving
- Proving

**using**: specified 'knowledge bases'

**applying**: simplifiers, solvers and provers from the *Theorema* library

- Composing
- Structuring mathematical texts
- Manipulating

**Program Verification in Theorema**

- **'Theorema Language'**: higher order predicate logic (with equalities as rewrite rules)  
  **verification**: proving specifications based on definitions (both are logical formulae)  
  (B. Buchberger, A. Craciun, N. Popov, T. Jebelean)

  Verification of Functional Programs using fix-point induction (N. Popov)

  Forward Verification for Recursive Programs (T. Jebelean)

- additionally: imperative language with interpreter and verifier

  **verification**: generating verification conditions  
  (L. Kovács)

**Advantages of Program Verification in Theorema**

- proofs in natural language and using natural style inferences
- access to powerful computing and solving algorithms (Mathematica)
Imperative Program Verification in Theorema

**Infrastructure** (based on work of M. Kirchner)

\(\textit{Special Constructs}\)

Program, Specification, Execute

\(\textit{Procedural Language}\)

*Example: Division of two natural numbers*

```
Specification["Division",
    Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo],
    Pre \rightarrow ((x \geq 0) \land (y > 0)),
    Post \rightarrow ((quo \ast y + rem = x) \land (0 \leq rem \land rem < y))]
```

```
Program["Division",
    Div[\downarrow x, \downarrow y, \uparrow rem, \uparrow quo],
    quo := 0;
    rem := x;
    WHILE[y \leq rem,
        rem := rem - y;
        quo := quo + 1,
        Invariant \rightarrow ((quo \ast y + rem = x) \land (0 \leq rem) \land (0 < y)
    TerminationTerm \rightarrow rem],
    Specification \rightarrow Specification["Division"]]
```

\(\textit{Interpreter}\)

*Example: executing the above program*

```
Execute[Div[57, 21, rm, qt]]; {rm, qt}
```

\{15, 2\}
\section*{Verification Condition Generator}

Example: generating the verification conditions for the above program

\begin{verbatim}
VCG[Program["Division"], Specification["Division"]]

Lemma (Division):
for any: x, y, rem, quo

(WHILE.Inv + Term) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land y \leq rem \land (T1 = rem)\)

(WHILE.Final) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (y \neq rem) \Rightarrow (rem + quo \ast y = x)\)

(WHILE.Term) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land y \leq rem \Rightarrow rem \geq 0\)

(Init) \(x \geq 0 \land y > 0 \Rightarrow (x = x) \land 0 \leq x \land 0 < y\)

Prove[Lemma["Division"], by \rightarrow PCS];

PCS: \((WHILE.Inv+Term)\) \((WHILE.Final)\) \((WHILE.Term)\) \((Init)\)
\end{verbatim}

\subsection*{Approach: Hoare Logic, Weakest Precondition}

\begin{verbatim}
{P}
\leftarrow wp_n
s_1
\leftarrow wp_{n-1}
s_2
\vdots
\leftarrow wp_1
s_n
{Q}

P \Rightarrow wp_n
\end{verbatim}
Specific Problems

\(\n\blacktriangledown\text{Loop Termination (handling termination terms)}\)

\(\blacktriangledown\text{Function Calls}\)

Using Function Call Maximum

```
Specification["Max", \(m = \text{Max}[\downarrow x, \downarrow y],\)]
\quad \text{Pre} \rightarrow (\text{IsInteger}[x] \land \text{IsInteger}[y]), \\
\quad \text{Post} \rightarrow (m = x \land x \geq y) \lor (m = y \land y > x)
```

```
Specification["Calculus", Calc[\downarrow a, \downarrow b, \downarrow y, \uparrow x], \\
\quad \text{Pre} \rightarrow (\text{IsInteger}[a] \land \text{IsInteger}[b]), \\
\quad \text{Post} \rightarrow ((x \geq (y + a)) \land (x \geq (y + b)))
```

```
Program["Calculus", Calc[\downarrow a, \downarrow b, \downarrow y, \uparrow x], \\
\quad x := y + \text{Max}[a, b]
```

(*simple version*)

```
VCG[Program["Calculus"], Specification["Calculus"]]
```

Lemma (Calculus):

for any \(a, b, y, x\)

(Init) \(\text{IsInteger}[a] \land \text{IsInteger}[b] \Rightarrow y + \text{Max}[a, b] \geq y + a \land y + \text{Max}[a, b] \geq y + b\)

(*developed version*)

```
VCG[Program["Calculus"], Specification["Calculus"]]
```

Lemma (Calculus):

for any \(a, b, y, x\)

(Init) \(\text{IsInteger}[a] \land \text{IsInteger}[b] \Rightarrow (\text{IsInteger}[a] \land \text{IsInteger}[b]) \land ((x1 = a \land a \geq b) \lor \)
**Execution**

**Automated Generation of Loop Assertions**
**Generation of Invariants**

Two approaches:
- static
- dynamic

techniques for discovering invariants.

**Dynamic**

Execute a program on a collection of inputs and infer invariants from captured variable traces.

The accuracy of the inferred invariant depends in part on the quality and completeness of the test cases; additional test cases might provide new data from which more accurate invariants can be inferred.

**DAIKON [ECGN00]**

- prototype invariant detector that implements a set of techniques for discovering invariants from execution

Detection process:
- instrumenting the source program to trace the variables of interest
- running the instrumented program over a set of test cases
- inferring invariants - over the instrumented variables
  - over derived variables that are not manifest in the original program.

**Andrews [Andrews98]**

compares actual behaviour against of a user-defined model, indicating divergences between the two

**Cook, Wolf [CW98]**

statistical techniques to detect sequencing, conditionals, iteration

**Verisoft**

finite state machines

**Static [Hoare69, Floyd77, Dijkstra76, Gries81]**

- static analysis operate on the program text, not on test runs
- reported properties are true for any program run
- theoretically, they can detect sound invariants
**Iterative Technique under the framework of abstract interpretation [Cousot77]**

- An invariant is true for every reachable state ⇒ over-approximation of the set of reachable states.

- Generating these over-approximations (iterative technique):
  - [Cousot78], [Karr76], [Bensalem96], [Manna97], [Tiwari01] - specially for Linear Invariants

  - start from the initial state
  - iterates until no more states are added
  - widening is used (for convergence): guess informally the limit of a sequence of iterative
  - widening might produce over-approximations that are too large to be useful

  These techniques are based upon [EGLW72]: *Difference Equation Method*, which proceeds

  1) by means of recurrence equations (called difference equations), explicit expression is found (as a function of the number of loop iteration k, other variables that remain constant in the loop and the input variables)

  2) k is eliminated to obtain invariant predicates

- Few results: Non-Linear Invariant Generation involving multiplication [Muller02]

**Constraint-based Technique**

- [COLON03]: "Linear Invariant Generation using Non-Linear Constraint Solving"

  - fix a template candidate assertion
  - generate constraints on the coefficients in order to guarantee its inductiveness
  - any solution → inductive assertion (invariant)
  - widening is not used
  - can generate stronger invariants than the iterative technique
  - its usage, in practice, limited to linear invariants of linear programs

**usage of Groebner Bases**

**FORWARD PROPAGATION**

- [Manna04]: "Non-linear Loop Invariant Generation using Groebner Bases"

  - generate algebraic inductive assertions of the form \( p(x_1, x_2, \ldots, x_n) = 0 \)
    \( (p \in \mathbb{R}[x_1, \ldots, x_n]) \)
- invariant generation problem → constraint solving problem
  (using properties describing the consequences of assertions)
- Groebner Bases are used to reduce the invariant generation problem to a non-linear constraint
- techniques for solving constraints → solution: Invariant

[Rodriguez-Carbonell03]: "Automatic Generation of Polynomial Loop Invariants for Imperative Programs with Conditional Statements and Assignments"
- finding invariants of loops in imperative programs with conditional statements and assignments
- generic procedure based on forward propagation for computing loop invariants as fixed points
  ⇒ applied to programs in which loop invariants can be formulated as polynomial equations:
  - for invertible assignments, whose powers have polynomial structures, polynomial loop invariants
  - termination of the procedure is guaranteed in \((2m+1)\) iterations
    \(m\) is the number of variables which are changing during the loop

Implementation: in Maple

\[ \text{\textbf{BACKWARD PROPAGATION}} \]

[MullerSeidl02]: "Polynomial Constants are Decidable"
- non-linear programs without branch-conditions
- avoid widening/narrowing, by observing that ideals in certain polynomial rings (Noetherian Rings)
  - exact technique for polynomial constants, i.e. invariants of the form \(x=p\)
    \(x=\text{program variable}, p=\text{polynomial in the program variables}\)

[MullerSeidl03]: "Computing Polynomial Program Invariants"
- extend [MullerSeidl02] that considered only detection of constants, by:
  \begin{itemize}
    \item checking validity of arbitrary polynomial relations: \(p(x_1, \ldots, x_k) = 0\) (and not just of the form 
      \(p\) a multi-variate polynomial in the program variables \(x_1, \ldots, x_k\))
    \item treating polynomial non-equality guards
    \item derive (and not just check) valid polynomial relations of bounded degree
  \end{itemize}
- For the checking algorithm (first step):
  compute polynomial ideal that represent the weakest precondition for the validity of the
  Rely on \textit{Hilbert's Basis Theorem & Buchberger's Algorithm}
- For the derivation:
  compute the weakest precondition of a generic polynomial relation at the target program.

∇ Our Approach: Combinatorial and algebraic methods
\textbf{Generation of Loop Invariants by Combinatorial Methods}

**One Loop**

\textbf{1st order recurrences}

\textit{GOSPER-Summable Recurrences}

Example: Integer division - with \texttt{WHILE} loop-

\begin{verbatim}
quo := 0;
rem := x;
WHILE [y ≤ rem,
   rem := rem - y;
   quo := quo + 1]
\end{verbatim}

\textbf{1st STEP}

We obtain the following recursion equations:

\begin{verbatim}
quo_0 = 0                     rem_0 = x
q_{uo_0} = q_{uo_0} + 1      r_{em_0} := r_{em_0} - y
\end{verbatim}

Explicit equations (by using combinatorics):

- \textbf{Gosper Algorithm} ( Gosper[summand, \{counter,LB,UB\} ] )
  RISC combinatorics group

\begin{verbatim}
quo_k = k                     rem_k = x - k \times y
\end{verbatim}

Eliminate \( k \):

\begin{verbatim}
rem = x - quo \times y
\end{verbatim}
2nd STEP - additional formulae

PostCondition of Loop:

\[(\text{quo} \times \text{y} + \text{rem} = \text{x}) \land (0 \leq \text{rem}) \land (\text{rem} < \text{y}) \land (0 < \text{y})\]

extracting relevant formulae, analysing Loop Condition

\[(\text{quo} \times \text{y} + \text{rem} = \text{x}) \land (0 \leq \text{rem}) \land (0 < \text{y})\]

\(\downarrow\)

\[(\text{quo} \times \text{y} + \text{rem} = \text{x}) \land (0 \leq \text{rem}) \land (0 < \text{y})\]

loop – critical variables : \{\text{quo}, \text{rem}\}
conditional – termination critical variables : \{\text{rem}, \text{y}\}
extracting relevant formulae, analysing critical variables

Examples

Invariant from Independent Recursive Equations

Example: Division
(WHILE with Aut. Generation of Invariant and given TerminationTerm)

Specification

"Division",
Div[\downarrow \text{x}, \downarrow \text{y}, \uparrow \text{rem}, \uparrow \text{quo}]
Pre \rightarrow (\text{Pre})
Post \rightarrow (\text{Post})
Program["Division",
   Div[↓x, ↓y, ↑rem, ↑quo],
   quo := 0;
   rem := x;
   WHILE[(y ≤ rem),
      rem := rem - y;
      quo := quo + 1,
      TerminationTerm → rem],
   Specification → Specification["Division"]]

\( ∇ \) Verification Condition Generator

VCG[Program["Division"]]

Lemma (Division):
   for any : x, y, rem, quo
   (WHILE.Inv + Term) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (rem + (-1) \ast x + quo \ast y \land
   (rem + (-1) \ast y + (1 + quo) \ast y = x) \land 0 \leq rem + (-1) \ast y \land 0 < y \land (rem + (-1) \ast x + (-
   (WHILE.Final) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (rem + (-1) \ast x + quo \ast y = 0) \land
   (WHILE.Term) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (rem + (-1) \ast x + quo \ast y = 0) /
   (Init) \(0 \leq x \land 0 < y \Rightarrow (x = x) \land 0 \leq x \land 0 < y \land (0 = 0)\)

Prove[Lemma["Division"], by → PCS];

PCS: (WHILE.Inv+Term) (WHILE.Final) (WHILE.Term) (Init)

Invariant from Dependent Recursive Equations

\( ∘ \) Example: Square Root (WHILE)

\( ∇ \) The Source Code and VCs of Integer Square Root

Specification["Integer Square Root", IntRootSpec[↓n, ↑k],
   Pre → (n ≥ 0),
   Post → (k ≥ 0 \land k^2 ≤ n ≤ (k + 1)^2)]
Program["Integer Square Root", IntRoot[\[\text{n}, \uparrow \text{k}\]],
Module[{j, m},
  k := 0;
  j := 1;
  m := 1;
  n := 0;
  WHILE[m \leq n,
    k := k + 1;
    j := j + 2;
    m := m + j,
    TerminationTerm \rightarrow (n - m)
  ]
]
]

VCG[Program["Integer Square Root"], Specification["Integer S<

Lemma (Integer Square Root):

for any \(n, k, j, m\)

(WHILE.Inv + Term) \((j = 1 + 2 \cdot k) \land (m = 1 + 2 \cdot k + k^2) \land k \geq 0 \land k^2 \leq n \leq (1 + k)(2 + j = 1 + 2 \cdot (1 + k)) \land (2 + j + m = 3 + 2 \cdot k + (1 + k)^2) \land -2 + (-1) \cdot j + (-1) \cdot m\)

(WHILE.Final) \((j = 1 + 2 \cdot k) \land (m = 1 + 2 \cdot k + k^2) \land k \geq 0 \land k^2 \leq n \leq (1 + k)^2 \land (m\)

(WHILE.Term) \((j = 1 + 2 \cdot k) \land (m = 1 + 2 \cdot k + k^2) \land k \geq 0 \land k^2 \leq n \leq (1 + k)^2 \land m\)

(Init) \(n \geq 0 \Rightarrow (1 = 1) \land (1 = 1)\)

Prove[Lemma["Integer Square Root"], by \rightarrow PCS]

PCS: \(\text{(WHILE.Inv+Term)} \quad \text{(WHILE.Final)} \quad \text{(WHILE.Term)} \quad \text{(Init)}\)

Example: Sum of Integers

The source code and the VCs

Specification["Sum", SimpleSum[\[\text{n}, \uparrow \text{x}\]], Pre \rightarrow (n \geq 1), Post \rightarrow ()]
Program["Sum", SimpleSum[↓n, ↑x], x := 0;
    FOR[i, 1, n,
        x := x + i
        (*, Invariant→(x=((i-1)*i)/2)*)
    ]]

VCG[Program["Sum"], Specification["Sum"]]

Lemma (Sum):
for any : n, x
(FOR.Inv1) \( (2 \times x = -1 \times i + i^2) \land 1 \leq i \land i \leq n \Rightarrow 2 \times (i + x) = -1 + (-1) \times i + (1 + i)^2 \)
(FOR.Inv2) \( (2 \times x = -1 + (-1) \times n + (1 + n)^2) \Rightarrow \left( x = \frac{1}{2} \times n \times (1 + n) \right) \)
(Init) \( n \geq 1 \Rightarrow (0 = 0) \)

Prove[Lemma["Sum"], by → PCS]

{• ProofObject •, • ProofObject •, • ProofObject •}

▽ Old Version's VCs

VCG[Program["Sum"], Specification["Sum"]]

Lemma (Sum):
for any : n, x
(FOR.Inv1) \( \left( x = \frac{(i - 1) \times i}{2} \right) \land (1 \leq i \land i \leq n) \Rightarrow \left( x + i = \frac{(i + 1 - 1) \times (i + 1)}{2} \right) \)
(FOR.Inv2) \( \left( x = \frac{(n + 1 - 1) \times (n + 1)}{2} \right) \Rightarrow \left( x = n \times \frac{n + 1}{2} \right) \)
(Init) \( n \geq 1 \Rightarrow \left( 0 = \frac{(1 - 1) \times 1}{2} \right) \)

PCS: (For.Inv1) (For.Inv2) (Init)
Example: Square with FOR

The Source Code and the VCs

```plaintext
Specification["Square with FOR", SFOR[↓n, ↑s],
Pre → (n ≥ 1),
Post → (s = n * n)]
```

```plaintext
Program["Square with FOR", SFOR[↓n, ↑s],
s := 0;
    FOR[i, 1, n,
        s := s + (2*i - 1)
        (*, Invariant→(s=i*i - 2*i + 1)*)
    ],
    Specification→Specification["Square with FOR"]
]
```

VCG[Program["Square with FOR"], Specification["Square with FOR"]]

- Join::heads : Heads List and <<58>> at positions 1 and 2 are expected to
- =::shape : Lists {<<54>>, <<55>>} and <<56>>[s, {}, 1, i, -1 + n, {}] are not
- $RecursionLimit::reclim : Recursion depth of 256 exceeded.
- $RecursionLimit::reclim : Recursion depth of 256 exceeded.
- $RecursionLimit::reclim : Recursion depth of 256 exceeded.
- General::stop : Further output of $RecursionLimit::reclim will be suppressed
- Join::heads : Heads Hold and <<64>> at positions 1 and 2 are expected to
- Join::heads : Heads Hold and <<64>> at positions 1 and 2 are expected to
- Join::heads : Heads Hold and <<64>> at positions 1 and 2 are expected to
- General::stop : Further output of Join::heads will be suppressed during this
- =::shape : Lists {<<54>>, <<55>>} and <<56>>[Join[{-1}, <<58>>[2, i, s]], <<
- $RecursionLimit::reclim : Recursion depth of 256 exceeded.
- $RecursionLimit::reclim : Recursion depth of 256 exceeded.
- $RecursionLimit::reclim : Recursion depth of 256 exceeded.
- General::stop : Further output of $RecursionLimit::reclim will be suppressed
- Join::heads : Heads Hold and <<64>> at positions 1 and 2 are expected to
Lemma (Square with FOR):

for any \( n, s \):

\[
\textbf{If}(\text{FOR.Inv1}, \text{TM}\text{Implies}(\text{TM} \text{And}(\text{Eliminate}(\text{Join}[, \text{TM}\text{Equal}(i, \text{TM} \text{Plus}(1, \text{Theorema} \text{Experimental} \text{Verification} \text{VCG Private}'' \text{gfs}619). \text{var}(n)), \text{TM} \text{Le} \text{Equal} \text{Plus}(1, i), \text{TM} \text{Plus}(1, \text{Theorema Experimental Verification} \text{VCG Private}'' \text{gfs}619), \text{var}(n)), \text{TM}\text{Equ} \text{Equal} \text{Plus}(1, \text{var}(n), \text{TM}\text{Equ} \text{Equal} \text{Plus}(1, \text{var}(n), 1))))), \text{Theorema Experimental Verification} \text{VCG Private}'' \text{par}1\text{S495}, \{x2 \rightarrow 0\}, \text{T}
\]

\[
\textbf{If}(\text{Init}, \text{TM}\text{Implies}(\text{TM}\text{GreaterEqual}(\text{var}(n), 1)), \text{Eliminate}(\text{Join}[\text{TM}\text{Set}'' \text{Equal}(1, \text{TM} \text{Plus}(1, \text{Theorema} \text{Experimental} \text{Verification} \text{VCG Private}'' \text{par}1\text{S495), \text{TM} \text{Set}(\text{TM} \text{Set}(), \text{TM} \text{Set}(), \text{Theorema} \text{Experimental} \text{Verification} \text{VCG Private}'' \text{par}1\text{S495, \text{TM} \text{Set}()})}
\]

\[
\textbf{Prove}[\text{Lemma["Square with FOR"], by } \rightarrow \text{PCS}]
\]

\[
\text{PCS: } \text{(For.Inv1)} \quad \text{(For.Inv2)} \quad \text{(Init)}
\]
Example: Closest integer to the Cubic root

Specification[
"CubicRoot", CubicRoot[\[\downarrow a, \uparrow r\]],
Pre \rightarrow a \geq 0,
Post \rightarrow (r - \frac{1}{2})^3 < a \land (r + \frac{1}{2})^3 > a]

Program["CubicRoot", CubicRoot[\[\downarrow a, \uparrow r\]],
Module[{x, s},
   x := a;
   r := 1;
   s := 13 / 4;
   WHILE[(x - s > 0),
      x := x - s;
      s := s + 6 * r + 3;
      r := r + 1,
      TerminationTerm \rightarrow (x - s)
   ]
]
]

VCG[Program["CubicRoot"], Specification["CubicRoot"]]

Lemma (CubicRoot):
for any \( a, r, x, s \)

(\text{WHILE.Inv + Term}) \[ s = \frac{1}{4} + 3 * r^2 \land \left( 2 * x = \frac{1}{2} + 2 * a + \left( -\frac{3}{2} \right) * r + 3 * r^2 \right) \land \left( 3 + s + 6 * r = \frac{1}{4} + 3 * (1 + r)^2 \right) \land \left( 2 * (-1 * s + x) = -\frac{1}{2} + 2 * a + \frac{1}{2} * (-1 + (-1)) * r \right) \]

(\text{WHILE.Final}) \[ s = \frac{1}{4} + 3 * r^2 \land \left( 2 * x = \frac{1}{2} + 2 * a + \left( -\frac{3}{2} \right) * r + 3 * r^2 + (-2) \right) \]

(\text{WHILE.Term}) \[ s = \frac{1}{4} + 3 * r^2 \land \left( 2 * x = \frac{1}{2} + 2 * a + \left( -\frac{3}{2} \right) * r + 3 * r^2 + (-2) \right) \]

(Init) \[ a \geq 0 \Rightarrow \left( \frac{13}{4} = \frac{13}{4} \right) \land (2 * a = 2 * a) \]
(eq) INVARIANT ≡ \((s = \frac{1}{4} + 3 \cdot r^2) \wedge (2 \cdot x = \frac{1}{2} + 2 \cdot a + (\frac{-3}{2}) \cdot r + 3 \cdot r^2)\)

- Non-Linear Recurrences of the form: \(x_n = t \cdot x_{n-1}\)

\(\nabla\) Mutual recurrences

*Example: Fibonacci Numbers*

**Specification**

```
Specification["Fibonacci", FibonacciProcSpec[↓n, ↑F], Pre→
```

**Program**

```
Program["Fibonacci", FibonacciProc[↓n, ↑F],
Module[{H, i},
  i := n;
  F := 1;
  H := 1;
  WHILE[i > 1,
    H := H + F;
    F := H - F;
    i := i - 1,
    TerminationTerm → i - 1]]
```

We obtain the following recursive equations:

- \(H_1 (= 1)\)
- \(F_1 (= 1)\)

- \(H_k = H_{k-1} + F_{k-1} \ (k \geq 1)\)
- \(F_k = H_k + F_{k-1} \ (k \geq 1)\)

- \(i_k = i_{k-1} - 1\)
- \(i_1 (= n)\)

\(k\) is the iteration index

- By Gosper:
  - \(i_k = n - (k - 1)\)

- By Generating Functions:
\[ F_k = \frac{\phi^k - \phi^{-k}}{\sqrt{5}} \]

\[ H_k = \frac{\phi^{k+1} - \phi^{k+1}}{\sqrt{5}} \]

\[
\text{where: } \phi = \frac{1 + \sqrt{5}}{2}
\]

Eliminate \( k \):

\[
\text{Invariant} \equiv \left( F = \frac{\phi^{n-1} - \phi^{-n-1}}{\sqrt{5}} \right) \wedge \left( H = \frac{\phi^{n-2} - \phi^{-n-2}}{\sqrt{5}} \right)
\]

Using Generating Functions [Stanley, 1980]

1. Recurrence equation for all \( n \in \mathbb{Z} \):

\[
H_k = H_{k-1} + F_{k-1} + H_{ini} \quad F_k = H_k + F_{k-1} + F_{ini}
\]

where \( (H_0 = H_{\text{negative}} = F_0 = F_{\text{negative}} = 0) \):

\[
H_{ini} = H_1 \quad F_{ini} = F_1 - H_1
\]

2. Switch to Generating Functions - by summation:

\[
F(z) = \sum_k F_k z^k \\
H(z) = \sum_k H_k z^k
\]

\[
\downarrow
\]

Harmonic Forms:

\[
H(z) = z H(z) + z F(z) + H_{ini} z
\]

\[
F(z) = H(z) - z F(z) + F_{ini} z
\]

\[
\downarrow
\]

Solving a system of 2 equations in 2 unknowns:
\[ F(z) = -\frac{z}{-1 + z + z^2} \]

(is in fact the well known Fibonacci generating function)

\[ H(z) = -\frac{z(1+z)}{-1 + z + z^2} \]

\[ \Downarrow \]

**Closed–form of the coefficients (Rational Expansion Theorem for Distinct Roots):**

\[ F_n = \frac{\phi^n - \bar{\phi}^n}{\sqrt{5}} \quad \Rightarrow n^{th} \text{ Fibonacci number} \]

\[ H_n = \frac{\phi^{n+1} - \bar{\phi}^{n+1}}{\sqrt{5}} \quad \Rightarrow (n+1)^{th} \text{ Fibonacci number} \]

\[ \left( \text{where: } \phi = \frac{1 + \sqrt{5}}{2} \right) \]

**Verification Conditions**
Lemma (Fibonacci):
for any \( n, F, H, i \)

\[
(W\text{ILE.Inv} + \text{Term}) \quad H = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{1+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -3 + 5 \frac{1}{2} \right) + 2^{-1+i*(-1)*n} *
\]

\[
F = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{1+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -1 + 5 \frac{1}{2} \right) + 2^{-1+i*(-1)*n} *
\]

\[
H + F = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{2+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -3 + 5 \frac{1}{2} \right) + 2 * \left( 1 + 5 \frac{1}{2} \right)
\]

\[
H = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{2+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -1 + 5 \frac{1}{2} \right) + 2^{-2+i*(-1)*n} *
\]

\[
(W\text{ILE.Final}) \quad H = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{1+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -3 + 5 \frac{1}{2} \right) + 2 * \left( 1 + 5 \frac{1}{2} \right)
\]

\[
F = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{1+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -1 + 5 \frac{1}{2} \right) + 2^{-1+i*(-1)*n} *
\]

\[
(W\text{ILE.Term}) \quad H = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{1+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -3 + 5 \frac{1}{2} \right) + 2 * \left( 1 + 5 \frac{1}{2} \right)
\]

\[
F = \left( \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) \right)^{1+i*n} * \left( -5 + 5 \frac{1}{2} \right) * \left( -1 + 5 \frac{1}{2} \right) + 2^{-1+i*(-1)*n} *
\]

\[
(\text{Init}) \quad n > 0 \Rightarrow \left( 1 = \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) * \left( -5 + 5 \frac{1}{2} \right) * \left( -3 + 5 \frac{1}{2} \right) + \frac{1}{2} * \left( 1 + 5 \frac{1}{2} \right) \right)
\]

\[
\left( 1 = \frac{1}{2} * \left( 1 + \left( -1 \right) * 5 \frac{1}{2} \right) * \left( -5 + 5 \frac{1}{2} \right) * \left( -1 + 5 \frac{1}{2} \right) + \frac{1}{2} * \left( 1 + 5 \frac{1}{2} \right) * \left( 5 + 5 \frac{1}{2} \right) \right)
\]

\[\nabla\text{ While Loops (with Verification of Termination)}\]

\{P\} while C do S endwhile \{Q\},

where S is a program and C is a predicate

if:

\[
(WT1) \quad P \Rightarrow I
\]

\[
(WT2) \quad (I \land C) \Rightarrow T \in \mathbb{N}
\]

(or \( T \in \mathbb{D} \), where \( \mathbb{D} \) is any domain with a relation of order denoted by \( < \) )

\[
(WT3) \quad \{I \land C \land (T= T_1)\} S \{I \land (T < T_1)\}
\]

\[
(WT4) \quad (I \land \neg C) \Rightarrow Q.
\]
where I is a Loop-Invariant, T is a Termination Term and $T_1$ is a new variable.

**remark:** These verification rules allow us to prove the termination of the program.

**Example:**

```plaintext
\{r = x \land q = 0\}
while (y \leq r) do
    r := r - y;
    q := q + 1
endwhile
\{r + y \ast q = x \land r < y\}
```

where we choose as:

- Termination term: $T \equiv r-y$
- Loop-Invariant: $I \equiv r + y \ast q = x$

produces the following verification conditions:

**(WT1)** $(r = x \land q = 0) \Rightarrow (r + y \ast q = x)$

**(WT2)** $((r + y \ast q = x) \land (y \leq r)) \Rightarrow (r - y) \in \mathbb{N}$

**(WT3)** $\{(r + y \ast q = x) \land (y \leq r) \land (r - y) = T_1\}$
    
    $r := r - y;$
    
    $q := q + 1$

    $\{(r + y \ast q = x) \land (r - y) < T_1\}$

**(WT4)** $((r + y \ast q = x) \land (y \leq r)) \Rightarrow (r + y \ast q = x \land r < y)$

**remark:** in the case of $y=0$, the proof of (WT3) will fail $\Rightarrow$ non-termination

**Nested Loops**

*Example BinaryPower (Two WHILEs)*
Specification["BinaryPower", BinPowerSpec[\(x, y, z\)],
  Pre \(\rightarrow (y \geq 0 \land \text{IsInteger}[y])\),
  Post \(\rightarrow (z = x^y)\)]

Program["BinaryPower", BinPower[\(x, y, z\)],
  Module[{a, b},
    IF[y == 0,
      z := 1,
      a := x;
      b := y;
      z := 1;

      \(< j >\) WHILE[b > 0,
        b := b - 1;
        z := z * a;

      \(< j_1 >\) WHILE[\text{IsEven}[b] \land b > 0,
        a := a * a;
        b := b / 2,
        TerminationTerm \rightarrow b
      ]
    ]
    TerminationTerm \rightarrow b
  ]
],
Specification \rightarrow Specification["BinaryPower"]
]

\textbf{Recurrence Solving}

Set of Loop-Variables: \(V = \{a, b, z\}\)

\textbf{Inner WHILE}

Detected recurrences:

\[
\begin{align*}
  b_{j_1+1} &= b_{j_1} / 2; \\
  a_{j_1+1} &= a_{j_1} \ast a_{j_1},
\end{align*}
\]

Explicit Forms
\[ b_{j_1} = b_{j_1 \text{ini}} / 2^{j_1} \]

\[ a_{j_1} = (a_{j_1 \text{ini}})^{2^{j_1}} \]

Eliminating \( j_1 \), and obtain the system on \( V \):

\[
\begin{align*}
(1) \quad (a_{j_1})^{b_{j_1}} &= (a_{j_1 \text{ini}})^{b_{j_1 \text{ini}}} \\
    z_{j_1} &= z_{j_1 \text{ini}}
\end{align*}
\]

- **Outer WHILE**

\[
\begin{align*}
(2) \quad b_j &= b_{j-1} - 1 \\
    z_j &= z_{j-1} \ast a_{j-1} \\
    a_j &= a_{j-1}
\end{align*}
\]

- **Relations between Loop Counters and Variables**

\[
\begin{align*}
(3) \quad a_{j_1} &\rightarrow a_{j+1} \quad a_{j_1 \text{ini}} &\rightarrow a_j \\
    b_{j_1} &\rightarrow b_{j+1} \quad b_{j_1 \text{ini}} &\rightarrow b_j \\
    z_{j_1} &\rightarrow z_{j+1} \quad z_{j_1 \text{ini}} &\rightarrow z_j
\end{align*}
\]

- **Invariant Generation**

From (1) & (2) & (3)

\[
\begin{align*}
(4) \quad b_j &= b_{j-1} - 1 \\
    z_j &= z_{j-1} \ast a_{j-1} \\
    a_j &= a_{j-1} \\
    a_{j+1}^{b_{j+1}} &= a_j^{b_j} \\
    z_{j+1} &= z_j
\end{align*}
\]

Eliminating the \( j^{th} \)-indexed variables:
Subgoal: Establish a formula $l[a, b, z]$ s.t. $l[a_{j+1}, b_{j+1}, z_{j+1}] = l[a_{j-1}, b_{j-1}, z_{j-1}]$

This is obtained by (heuristics) multiplying parts of (5):

$z_{j0} \cdot a_{j0} \cdot b_{j0} = z_{j0-1} \cdot a_{j0-1} \cdot b_{j0-1}$

Hence, by substituting the initial values:

$z \cdot a^b = x^y$
Diamond Generation of Termination Term

Example: Integer division - with WHILE loop-

\[
\begin{align*}
\text{quo} &:= 0; \\
\text{rem} &:= x; \\
\text{WHILE}[y \leq \text{rem}, \\
\quad \text{rem} := \text{rem} - y; \\
\quad \text{quo} := \text{quo} + 1]
\end{align*}
\]

From Loop Condition, we obtain:

\[
\text{TerminationTerm} \equiv \text{rem} - y
\]

Other situations:

- Conjunction in Condition: TerminationTerm = Product of expressions
- Disjunction in Condition: TerminationTerm = Maximum of expressions

Examples

Invariant from Independent Recursive Equations

- Division (WHILE with Aut. Generation of Invariant and TerminationTerm)

Specification:

\[
\begin{align*}
\text{Specification} &= \text{"Division"}, \\
\quad \text{Div}[@x, @y, ↑\text{rem}, ↑\text{quo}], \\
\quad \text{Pre} \rightarrow ((0 \leq x) \land (0 < y)), \\
\quad \text{Post} \rightarrow ((\text{quo} \cdot y + \text{rem} = x) \land (0 \leq \text{rem}) \land (\text{rem} < y)),
\end{align*}
\]

Program["Division",
    Div[↓x, ↓y, ↑rem, ↑quo],
    quo := 0;
    rem := x;
    WHILE[(y ≤ rem),
        rem := rem − y;
        quo := quo + 1],
    Specification → Specification["Division"]]

\[ \text{Verification Condition Generator} \]

VCG[Program["Division"]]

Lemma (Division):
    for any : x, y, rem, quo
    (WHILE.Inv + Term) \((rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (rem + (-1) \ast x + quo \ast y = (rem + (-1) \ast y + (1 + quo) \ast y = x) \land 0 \leq rem + (-1) \ast y \land 0 < y \land (rem + (-1) \ast x - (rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (rem + (-1) \ast x + quo \ast y = (rem + quo \ast y = x) \land 0 \leq rem \land 0 < y \land (rem + (-1) \ast x + quo \ast y = (Init) \quad 0 \leq x \land 0 < y \Rightarrow (x = x) \land 0 \leq x \land 0 < y \land (0 = 0) \]

\[ \text{Prove[Lemma["Division"], by → PCS]} \]

\{- ProofObject -, - ProofObject -, - ProofObject -, - ProofObject\}

PCS: (WHILE.Inv+Term) (WHILE.Final) (WHILE.Term) (Init)

Invariant from Dependent Recursive Equations

\[ \text{Example: Square Root} \]

(WHILE with Aut. Generation of Invariant and TerminationTerm)

\[ \text{The source code and its specification; Verification Conditions} \]

Specification["Integer Square Root", IntRootSpec[↓n, ↑k],
    Pre → ((n ≥ 0)),
    Post → (k ≥ 0 \land k^2 \leq n \leq (k + 1)^2)\]
Program "Integer Square Root", IntRoot[↓n, ↑k],
Module[{j, m},
  k := 0;
  j := 1;
  m := 1;
  n := 0;
  WHILE[m ≤ n,
    k := k + 1;
    j := j + 2;
    m := m + j
  ]
]
]

VCG[Program "Integer Square Root"], Specification "Integer :

Lemma ( Integer Square Root ):
  for any : n, k, j, m
  (WHILE.Inv + Term)    k ≥ 0 ∧ k^2 ≤ n ≤ (1 + k)^2 ∧ (j = 1 + 2 * k) ∧ (m = 1 + k + 2 * j + 2 * (1 + k)) ∧ (2 + j + m = 2 + k).
  (WHILE.Final)         k ≥ 0 ∧ k^2 ≤ n ≤ (1 + k)^2 ∧ (j = 1 + 2 * k) ∧ (m = 1 + k + 2 * k^2).
  (WHILE.Term)          k ≥ 0 ∧ k^2 ≤ n ≤ (1 + k)^2 ∧ (j = 1 + 2 * k) ∧ (m = 1 + k + 2 * k^2).
  (Init)                n ≥ 0 ⇒ (1 = 1) ∧ (1 = 1)

Prove[Lemma "Integer Square Root"], by → PCS

{- ProofObject -, - ProofObject -, - ProofObject -, - ProofObject -}

PCS:  (WHILE.Inv+Term)  (WHILE.Final)  (WHILE.Term)  (Init)
Conclusion and Future Work

★ Invariant Generation Problem by algebraic and combinatorial methods
  (summation, recurrence solving, variable elimination)

**Loops only with assignments:**

- efficient method for Single-While Loops Programs
  (Gosper, Generating Function)

- ongoing work on Nested While Loops

**Loops with conditionals**: future work (combination of closed-forms)

**Existing Approach**: Groebner Bases (Buchberger Algorithm)
  [Manna, Rodriguez, Müller – Olm]

★ Proof-analyzer

★ Handling linear inequalities

**Existing Approach for Termination**: Linear ranking functions
  [Podelski & Rybalchenko]

**Existing Approach for Invariant**: Non-linear constraint solving
  [Colon]

Quantifier Elimination
References

[AptO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog: Verification of Sequential and Concurrent


[ClarkSickel77] K.L. Clark and S.Sickel. Predicate logic: a calculus for the formal derivation of programs.


[Cousot78] P.Cousot and N.Balwachs. Automatic Discovery of Linear Restraints among the Variables


[ECGN00] Michael D. Ernst, Jake Cockrell, William G. Griswold and David Notkin. Dynamically

[Futschek89] Gerald Futschek. Programmentwicklung und Verifikation. Springer Verlag Wien No


