Generating Loop Invariants for Program Verification by Transformation

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
hamilton@computing.dcu.ie

VPT 2017

Outline

Introduction

- The verification of imperative programs generally involves annotating the program with assertions.
  - A theorem prover can be used to check these.
- Central to this annotation process is the use of loop invariants.
  - These are assertions that are true before and after each iteration of a loop.
- Finding these invariants is a difficult and time-consuming task.
  - Programmers are therefore reluctant to do this.
- We present a technique for automatically discovering loop invariants using program transformation.
  - Avoids the possible exponential blow-up in the size of the assertions produced by other techniques.

Language

Syntax

$S ::= \text{SKIP}$	Do nothing
$\mid V := E$	Assignment
$\mid S_1 \ ; \ S_2$	Sequence
$\mid \text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2$	Conditional
$\mid \text{BEGIN VAR } V_1 \ldots V_n \ S \text{ END}$	Local block
$\mid \text{WHILE } B \text{ DO } S$	While loop
$E ::= V$	Variable
$\mid C \ E_1 \ldots E_k$	Constructor Application
$\mid \lambda V.E$	λ-Abstraction
$\mid F$	Function Call
$\mid E_0 \ E_1$	Application
$\mid \textbf{case } E_0 \textbf{ of } P_1 \rightarrow E_1 \mid \cdots \mid P_k \rightarrow E_k$	Case Expression
$\mid E_0 \textbf{ where } F_1 = E_1 \ldots F_n = E_n$	Local Function Definitions

Language

E corresponds to natural number expressions which belong to the following datatype:

$$Nat ::= \text{Zero} \mid \text{Succ } Nat$$

B corresponds to boolean expressions which belong to the following datatype:

$$Bool ::= \text{True} \mid \text{False}$$

We assume a number of pre-defined operators written in this language; these definitions are unfolded and folded during transformation:

- Arithmetic operators: $+, -, *, /, \%, ^\wedge$
- Boolean operators: $\wedge, \vee, \neg, \Rightarrow$
- Relational operators: $<, >, \leq, \geq, =, \neq$

Partial Correctness

Floyd-Hoare Logic

$$\{P\}\ \text{SKIP}\ \{P\}$$

$$\{Q\{V := E\}\}\ V := E\ \{Q\}$$

$$\frac{\{P\}\ S_1\ \{Q\}, \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}}$$

$$\frac{\{P \wedge B\}\ S_1\ \{Q\}, \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \text{IF}\ B\ \text{THEN}\ S_1\ \text{ELSE}\ S_2\ \{Q\}}$$

$$\frac{\{P\}\ S\ \{Q\}, \quad V_1 \ldots V_n \notin fv(P), fv(Q)}{\{P\}\ \text{BEGIN VAR}\ V_1 \ldots V_n\ S\ \text{END}\ \{Q\}}$$

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \text{WHILE}\ B\ \text{DO}\ S\ \{I \wedge \neg B\}}$$

$$\frac{P \Rightarrow P', \quad \{P'\}\ S\ \{Q\}}{\{P\}\ S\ \{Q\}}$$

$$\frac{\{P\}\ S\ \{Q'\}, \quad Q' \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

Loop Invariants

- A loop invariant is an assertion that is true before and after each iteration of the loop.
- Usually needs to be provided by the programmer.
- A program which is annotated in this way will have the form: $\{P\}$ WHILE B DO $\{I\}$ S $\{Q\}$
- The three requirements of the invariant I are as follows:
  1. $P \Rightarrow I$
  2. $\{I \wedge B\}\ S\ \{I\}$
  3. $(I \wedge \neg B) \Rightarrow Q$

Example

```
{n ≥ 0}
x := 0;
y := 1;
WHILE x < n DO
   BEGIN
      x := x + 1;
      y := y * k
   END
{y = k^n}
```

Example

```
{n ≥ 0}
x := 0;
y := 1;
WHILE x < n DO
   BEGIN
      x := x + 1;
      y := y * k
   END
{y = k^n}
```

- Invariant is often a weakening of the postcondition.

Example

$\{n \geq 0\}$
$x := 0;$
$y := 1;$
WHILE $x < n$ DO
  BEGIN
    $x := x + 1;$
    $y := y * k$
  END
$\{y = k\hat{}n\}$

- Invariant is often a weakening of the postcondition.
- For example: $y = k\hat{}x$

Example

$\{n \geq 0\}$
x := 0;
y := 1;
WHILE x < n DO
  BEGIN
    x := x + 1;
    y := y ∗ k
  END
$\{y = k \wedge n\}$

- Invariant is often a weakening of the postcondition.
- For example: $y = k \wedge x$
- This does not satisfy the third requirement for an invariant:
  $y = k \wedge x \land \neg(x < n) \Rightarrow y = k \wedge n$

Example

$\{n \geq 0\}$
x := 0;
y := 1;
WHILE x < n DO
  BEGIN
    x := x + 1;
    y := y * k
  END
$\{y = k\wedge n\}$

- Invariant is often a weakening of the postcondition.
- For example: $y = k\wedge x$
- This does not satisfy the third requirement for an invariant: $y = k\wedge x \wedge \neg(x < n) \Rightarrow y = k\wedge n$
- The additional invariant $x \leq n$ is also required.

Weakest Liberal Precondition

- We make use of the weakest liberal precondition originally proposed by Dijkstra.
- This is denoted as $WLP(S,Q)$, where S is a program and Q is a postcondition.
- The condition $P = WLP(S,Q)$ if Q is true after execution of S, and no condition weaker than P satisfies this.
- The rules for calculating $WLP(S,Q)$ for our programming language are as follows:

$$
\begin{aligned}
WLP(\mathrm{SKIP},Q) &= Q \\
WLP(V := E, Q) &= Q\{V := E\} \\
WLP(S_1; S_2, Q) &= WLP(S_1, WLP(S_2, Q)) \\
WLP(\mathrm{IF}\ B\ \mathrm{THEN}\ S_1\ \mathrm{ELSE}\ S_2, Q) &= (B \Rightarrow WLP(S_1,Q)) \wedge (\neg B \Rightarrow WLP(S_2,Q)) \\
WLP(\mathrm{BEGIN\ VAR}\ V_1 \ldots V_n\ S\ \mathrm{END}, Q) &= WLP(S,Q),\ \text{where}\ V_1 \ldots V_n \notin fv(Q) \\
WLP(\mathrm{WHILE}\ B\ \mathrm{DO}\ \{I\}\ S, Q) &= I \wedge ((B \wedge I) \Rightarrow WLP(S,I)) \wedge ((\neg B \wedge I) \Rightarrow Q)
\end{aligned}
$$

Distillation

- We also make use of the distillation program transformation algorithm.
- Unfold/fold program transformation that builds on top of positive supercompilation and is strictly more powerful.
- Extra power is due to generalisation and folding being performed with respect to recursive terms.
- Terms are transformed into a normalised form called distilled form that makes it easier to identify similarities and differences between terms.
- Built-in associative operators (such as $+, *, \wedge, \vee$) are always transformed into right-associative form.

Embedding

- Generalisation is performed if the expression obtained from distillation is an embedding of a previously distilled one.
- The form of embedding we use is known as homeomorphic embedding.
- An expression E is embedded in expression E' if $E \trianglelefteq E'$.

$$\frac{}{V \trianglelefteq V'} \qquad \frac{\exists i \in \{1 \ldots n\}.E \trianglelefteq E_i}{E \trianglelefteq \phi(E_1, \ldots, E_n)} \qquad \frac{\forall i \in \{1 \ldots n\}.E_i \trianglelefteq E'_i}{\phi(E_1, \ldots, E_n) \trianglelefteq \phi(E'_1, \ldots, E'_n)}$$

- We write $E \preceq E'$ if expression E is coupled with expression E' using the third rule above.

Generalisation

- The generalisation of expression E with respect to expression E' (denoted by $E \sqcap E'$) is defined as follows:

$$E \sqcap E' = \begin{cases} (\phi(E''_1, \ldots, E''_n), \bigcup_{i=1}^{n} \theta_i, \bigcup_{i=1}^{n} \theta'_i), & \text{if } \phi = \phi' \\ \quad \text{where} & \\ \quad E = \phi(E_1, \ldots, E_n) & \\ \quad E' = \phi'(E'_1, \ldots, E'_n) & \\ \quad \forall i \in \{1 \ldots n\}. E_i \sqcap E'_i = (E''_i, \theta_i, \theta'_i) & \\ (V, \{V \mapsto E\}, \{V \mapsto E'\}), & \text{otherwise } (V \text{ is fresh}) \end{cases}$$

- The result of this generalisation is a triple (E'', θ, θ') where E'' is the generalised expression and θ and θ' are substitutions s.t. $E''\theta \equiv E$ and $E''\theta' \equiv E'$.

Most Specific Generalisation

- The most specific generalisation of expressions E and E' is an expression E'' such that for every other generalisation E''' of E and E', there is a substitution θ such that $E''\theta \equiv E'''$.
- The most specific generalisation of expressions E and E' (denoted by $E \triangle E'$) is computed by exhaustively applying the following rewrite rule to the triple obtained from the generalisation $E \sqcap E'$:

$$\left(\begin{array}{c} E, \\ \{V_1 \mapsto E', V_2 \mapsto E'\} \cup \theta, \\ \{V_1 \mapsto E'', V_2 \mapsto E''\} \cup \theta' \end{array}\right) \Rightarrow \left(\begin{array}{c} E\{V_1 \mapsto V_2\}, \\ \{V_2 \mapsto E'\} \cup \theta, \\ \{V_2 \mapsto E''\} \cup \theta' \end{array}\right)$$

The Induction-Iteration Method

- First proposed by Suzuki and Ishihata, 1977
- For the annotated program {P} WHILE B DO S {Q}, the logical assertion which is true if the loop is exited is calculated as follows:

$$P_0 = (\neg B \Rightarrow Q)$$

- Then, the weakest liberal precondition is used to calculate the logical assertion which is true before each execution of the loop body (in reverse order):

$$P_{i+1} = (B \Rightarrow WLP(S, P_i))$$

- The weakest liberal precondition of the loop is given by $\bigwedge_{i=0}^{\infty} P_i$.
- Successive approximations $I_j = \bigwedge_{i=0}^{j} P_i$ are calculated until one is found that is a loop invariant.

The Induction-Iteration Method

There are a few drawbacks to this approach:

- It is not guaranteed to terminate.
    - This is avoided by limiting the number of iterations.
    - It is found that in practice very few iterations are actually required.
- There can be an exponential blow-up in clauses into increasingly larger conjunctions.
    - This is particularly the case for conditionals, which double the number of possible paths through the loop body.
    - This is also a problem for other approaches that work forwards from the precondition using a strongest postcondition semantics.
- It still requires that the programmer provides the postcondition.
    - This is much less onerous than providing loop invariants and generally forms part of the specification of the program.

Our Approach

To avoid exponential blow-up in clauses:

- Conjuncts of clauses are simplified using distillation.
- Resulting conjuncts are combined using generalisation.
- Conjuncts for different paths through loop body are often minor variations of each other due to effects of distillation.

To ensure termination:

- If the current approximation is an embedding of a previous one then it is generalised with respect to this previous approximation.
- This process is continued until the current approximation is a renaming of a previous one; this is then the putative invariant for the loop.
- Guaranteed to terminate since embedding relation is a well-quasi order.

Our Approach

Our algorithm for the automatic generation of an invariant for the loop WHILE B DO S with postcondition Q is as follows:

$f\ (distill(\neg B \wedge Q))\ \emptyset$

where

$f\ P\ \phi =$ **if** $\exists Q \in \phi$ s.t. $Q \equiv P$ (modulo variable renaming)

then return P

else if $\exists Q \in \phi$ s.t. $Q \preceq P$

then $f\ P'\ \phi$ where $P' = P \triangle Q$

else return $f\ (\triangle_{i=1}^{n}\{distill(B \wedge P_i))\ (\phi \cup \{P\})$

where $WLP(S, P) = \bigwedge_{i=1}^{n} P_i$

Our Approach

The generated invariant may contain generalisation variables.

- We try to find values for these variables that satisfy each of the three requirements for loop invariants using our Poitín theorem prover (this could also be done using a SAT solver).
- For the annotated program $\{P\}$ WHILE B DO $\{I\}$ S $\{Q\}$, the initial value of variable v can be obtained by satisfying the following predicate for v_0 using the first requirement:

$$P \Rightarrow I\{v := v_0\}$$

- The inductive definition of v can be obtained by satisfying the following predicate for v_{i+1} using the second requirement:

$$I\{v := v_i\} \wedge B \Rightarrow WLP(S, I\{v := v_{i+1}\})$$

- The final value of v can be obtained by satisfying the following predicate for v_n using the third requirement:

$$(I\{v := v_n\} \wedge \neg B) \Rightarrow Q$$

Example

Consider again the previous example program:

$$
\begin{array}{l}
\{n \geq 0\} \\
x := 0; \\
y := 1; \\
\text{WHILE } x < n \text{ DO} \\
\quad \text{BEGIN} \\
\quad\quad x := x + 1; \\
\quad\quad y := y * k \\
\quad \text{END} \\
\{y = k^\wedge n\}
\end{array}
$$

Example

We calculate the logical assertion which is true if the loop is exited:

$$\neg(\mathsf{x} < \mathsf{n}) \wedge \mathsf{y} = \mathsf{k}^\wedge\mathsf{n}$$

This is simplified by distillation to the following:

$$\mathsf{x} \geq \mathsf{n} \wedge \mathsf{y} = \mathsf{k}^\wedge\mathsf{n} \tag{1}$$

Then, we calculate the logical assertion which is true before the final execution of the loop body:

$$\begin{array}{c} \mathsf{WLP}(\mathsf{BEGIN}\ \mathsf{x} := \mathsf{x} + 1; \mathsf{y} := \mathsf{y} * \mathsf{k}\ \mathsf{END}, \mathsf{x} \geq \mathsf{n} \wedge \mathsf{y} = \mathsf{k}^\wedge\mathsf{n}) \\ = \mathsf{x} + 1 \geq \mathsf{n} \wedge \mathsf{y} * \mathsf{k} = \mathsf{k}^\wedge\mathsf{n} \end{array}$$

In conjunction with the loop condition $(x < n)$, this is simplified to the following by distillation:

$$\mathsf{x} + 1 = \mathsf{n} \wedge \mathsf{y} * \mathsf{k} = \mathsf{k}^\wedge\mathsf{n} \tag{2}$$

This is not an embedding of (1), so the calculation continues.

Example

We next calculate the logical assertion which is true before the penultimate execution of the loop body:

$$\begin{aligned}&\mathrm{WLP}(\mathrm{BEGIN}\ \mathrm{x} := \mathrm{x}+1; \mathrm{y} := \mathrm{y} * \mathrm{k}\ \mathrm{END}, \mathrm{x}+1 = \mathrm{n} \wedge \mathrm{y} * \mathrm{k} = \mathrm{k}^\wedge\mathrm{n})\\ &= (\mathrm{x}+1)+1 = \mathrm{n} \wedge (\mathrm{y} * \mathrm{k}) * \mathrm{k} = \mathrm{k}^\wedge\mathrm{n}\end{aligned}$$

In conjunction with the loop condition ($x < n$), this is simplified to the following by distillation:

$$\mathrm{x}+2 = \mathrm{n} \wedge \mathrm{y} * (\mathrm{k} * \mathrm{k}) = \mathrm{k}^\wedge\mathrm{n} \tag{3}$$

We can see that (3) is an embedding of (2), so (3) is generalised to produce the following:

$$\mathrm{x}+\mathrm{v} = \mathrm{n} \wedge \mathrm{y} * \mathrm{w} = \mathrm{k}^\wedge\mathrm{n} \tag{4}$$

where v and w are new generalisation variables. This is not an embedding of (2) or (1), so the calculation continues.

Example

The logical assertion which is true before execution of the loop body is now re-calculated as follows:

$$\mathrm{WLP}(\mathrm{BEGIN}\ x := x+1; y := y * k\ \mathrm{END}, x+v = n \wedge y * w = k\hat{\ }n)$$
$$= (x+1)+v = n \wedge (y*k)*w = k\hat{\ }n$$

In conjunction with the loop condition $(x < n)$, this is simplified to the following by distillation:

$$x + (v+1) = n \wedge y * (k * w) = k\hat{\ }n \qquad (5)$$

We can see that (5) is an embedding of (4), so (5) is generalised to produce the following:

$$x + v' = n \wedge y * w' = k\hat{\ }n \qquad (6)$$

where v' and w' are new generalisation variables. We can now see that (6) is a renaming of (4), so (6) is our putative invariant.

Example

- We now try to find inductive definitions for the generalisation variables v' and w' from the three requirements of loop invariants using our theorem prover Poitín.
- The initial values of the generalisation variables (v'_0 and w'_0) can be determined using the first invariant requirement:

$$\mathrm{n} \geq 0 \wedge \mathrm{x} = 0 \wedge \mathrm{y} = 1 \Rightarrow \mathrm{x} + \mathrm{v}'_0 = \mathrm{n} \wedge \mathrm{y} * \mathrm{w}'_0 = \mathrm{k}^\wedge\mathrm{n}$$

- The assignments $v'_0 := n$ and $w'_0 := k^\wedge n$ satisfy this assertion.
- The inductive values of the generalisation variables (v'_{i+1} and w'_{i+1}) can be determined using the second invariant requirement:

$$\mathrm{x} + \mathrm{v}'_\mathrm{i} = \mathrm{n} \wedge \mathrm{y} * \mathrm{w}'_\mathrm{i} = \mathrm{k}^\wedge\mathrm{n} \wedge \mathrm{x} < \mathrm{n} \Rightarrow$$
$$(\mathrm{x}+1) + \mathrm{v}'_{\mathrm{i}+1} = \mathrm{n} \wedge (\mathrm{y} * \mathrm{k}) * \mathrm{w}'_{\mathrm{i}+1} = \mathrm{k}^\wedge\mathrm{n}$$

- The assignments $v'_{i+1} := v'_i - 1$ and $w'_{i+1} := w'_i / k$ satisfy this assertion.

Example

- The final values of the generalisation variables (v'_n and w'_n) can be determined using the third invariant requirement:

$$\mathsf{x} + \mathsf{v}'_\mathsf{n} = \mathsf{n} \wedge \mathsf{y} * \mathsf{w}'_\mathsf{n} = \mathsf{k}^\wedge\mathsf{n} \wedge \neg(\mathsf{x} < \mathsf{n}) \Rightarrow \mathsf{y} = \mathsf{k}^\wedge\mathsf{n}$$

- The assignments $v'_n := 0$ and $w'_n = 1$ satisfy this assertion.
- The discovered invariant is therefore equivalent to the following:

$$\mathsf{x} \leq \mathsf{n} \wedge \mathsf{y} = \mathsf{k}^\wedge\mathsf{x}$$

Example 2

Consider following program:

```
{n ≥ 0}
x := n;
y := 1;
z := k;
WHILE x > 0 DO
  BEGIN
    IF x%2 = 1 THEN y := y * z ELSE SKIP;
    x := x/2;
    z := z * z
  END
{y = k^n}
```

We use S to denote the body of the loop in the above program

Example 2

We calculate the logical assertion which is true if the loop is exited:

$$\neg(x > 0) \wedge y = k^\wedge n$$

This is simplified by distillation to the following:

$$x \leq 0 \wedge y = k^\wedge n \tag{1}$$

Then, we calculate the logical assertion which is true before the final execution of the loop body:

$$\mathrm{WLP}(S, x \leq 0 \wedge y = k^\wedge n)$$

This gives the following:

$$\begin{array}{c} (x\%2 = 1 \Rightarrow x/2 \leq 0 \wedge y * z = k^\wedge n) \\ \wedge\ (\neg(x\%2 = 1) \Rightarrow x/2 \leq 0 \wedge y = k^\wedge n) \end{array}$$

In conjunction with the loop condition ($x > 0$), the second conjunct is simplified to True by distillation.

Example 2

The first conjunct is simplified to the following:

$$x = 1 \wedge y * z = k^{\wedge} n \tag{2}$$

This is not an embedding of (1), so the calculation continues. We next calculate the logical assertion which is true before the penultimate execution of the loop body:

$$\begin{array}{c} \mathrm{WLP}(S, x = 1 \wedge y * z = k^{\wedge} n) \\ = (x\%2 = 1 \Rightarrow x/2 = 1 \wedge (y * z) * (z * z) = k^{\wedge} n) \\ \wedge\ (\neg(x\%2 = 1) \Rightarrow x/2 = 1 \wedge y * (z * z) = k^{\wedge} n) \end{array}$$

In conjunction with the loop condition ($x > 0$), the first conjunct is simplified to the following by distillation:

$$x = 3 \wedge y * (z * (z * z)) = k^{\wedge} n$$

and the second conjunct is simplified to the following:

$$x = 2 \wedge y * (z * z) = k^{\wedge} n$$

Example 2

These are generalised with respect to each other to give:

$$x = v \land y * (z * w) = k^\wedge n \qquad (3)$$

where v and w are new generalisation variables. This is not an embedding of (2) or (1), so the logical assertion which is true before execution of the loop body is now re-calculated as follows:

$$\begin{array}{c} \mathrm{WLP}(S, x = v \land y * (z * w) = k^\wedge n) \\ = (x\%2 = 1 \Rightarrow x/2 = v \land (y * z) * ((z * z) * w) = k^\wedge n) \\ \land\ (\neg(x\%2 = 1) \Rightarrow x/2 = v \land y * ((z * z) * w) = k^\wedge n) \end{array}$$

In conjunction with the loop condition ($x > 0$), the first conjunct is simplified to the following by distillation:

$$x = v * 2 + 1 \land y * (z * (z * (z * w))) = k^\wedge n$$

and the second conjunct is simplified to the following:

$$x = v * 2 \land y * (z * (z * w)) = k^\wedge n$$

Example 2

These are generalised with respect to each other to give:

$$x = v' \wedge y * (z * (z * w')) = k^\wedge n \tag{4}$$

where v' and w' are new generalisation variables. We can see that (4) is an embedding of (3), so (4) is generalised to give the following:

$$x = v'' \wedge y * (z * w'') = k^\wedge n \tag{5}$$

where v'' and w'' are new generalisation variables. We can now see that (5) is a renaming of (3), so (5) is our putative invariant.

Example 2

- We now try to find inductive definitions for the generalisation variables v'' and w'' from the three requirements of loop invariants using our theorem prover Poitín.
- The initial values of the generalisation variables (v''_0 and w''_0) can be determined using the first invariant requirement:

$$\mathrm{n} \geq 0 \wedge \mathrm{x} = \mathrm{n} \wedge \mathrm{y} = 1 \wedge \mathrm{z} = \mathrm{k} \Rightarrow \mathrm{x} = \mathrm{v}''_0 \wedge \mathrm{y} * (\mathrm{z} * \mathrm{w}''_0) = \mathrm{k}^\wedge\mathrm{n}$$

- The assignments $v''_0 := n$ and $w''_0 := k^\wedge(n-1)$ satisfy this assertion.
- The inductive values of the generalisation variables (v''_{i+1} and w''_{i+1}) can be determined using the second invariant requirement:

$$\begin{gathered}(\mathrm{x} = \mathrm{v}''_\mathrm{i} \wedge \mathrm{y} * (\mathrm{z} * \mathrm{w}''_\mathrm{i}) = \mathrm{k}^\wedge\mathrm{n} \wedge \mathrm{x} > 0) \Rightarrow \\ (\mathrm{x}\%2 = 1 \Rightarrow \mathrm{x}/2 = \mathrm{v}''_{\mathrm{i}+1} \wedge (\mathrm{y} * \mathrm{z}) * ((\mathrm{z} * \mathrm{z}) * \mathrm{w}''_{\mathrm{i}+1}) = \mathrm{k}^\wedge\mathrm{n}) \wedge \\ (\neg(\mathrm{x}\%2 = 1) \Rightarrow \mathrm{x}/2 = \mathrm{v}''_{\mathrm{i}+1} \wedge \mathrm{y} * ((\mathrm{z} * \mathrm{z}) * \mathrm{w}''_{\mathrm{i}+1}) = \mathrm{k}^\wedge\mathrm{n})\end{gathered}$$

Example 2

- The assignments $v''_{i+1} := v''_i/2$ and $(x\%2 = 1 \Rightarrow w''_{i+1} := w''_i/(z * z)) \wedge ((\neg(x\%2 = 1) \Rightarrow w''_{i+1} := w''_i/z)$ satisfy this assertion.
- The final values of the generalisation variables (v''_n and w''_n) can be determined using the third invariant requirement:

$$x = v''_n \wedge y * (z * w''_n) = k\hat{}n \wedge \neg(x > 0) \Rightarrow y = k\hat{}n$$

- The assignments $v''_n := 0$ and $w''_n = 1/z$ satisfy this assertion.
- The discovered invariant is therefore equivalent to the following:

$$x = n/2^j \wedge y = k\hat{}(n\%2^j)$$

Conclusions

We have described a technique for automatically discovering loop invariants.

- Similar to the induction-iteration method of Suzuki and Ishihata.
- Overcomes the problem of potential non-termination.
- Avoids the potential exponential blow-up in clauses into increasingly larger conjunctions.
- Still requires that the programmer provides the postcondition for the program.
- Of course, over-generalisation can occur, and a valid loop invariant not found.

Related Work

- Abstract interpretation:
  - Predicate abstraction: replace predicates with variables.
  - Constraint-based techniques over non-trivial mathematical domains (such as polynomials or convex polyhedra).
- Proof planning: Ireland and Stark, 1997
- Dynamic methods: Ernst et al., 2001
- Use of heuristics: Furia and Meyer, 2010
- Induction-iteration method: Suzuki and Ishihata, 1977

Further Work

- Extending for languages with richer features.
  - Unbounded data structures such as arrays: loop invariants need to be universally quantified.
  - Pointers: separation logic extends Floyd-Hoare logic to be able to handle pointers.
- Extending to reason about termination.
  - Using the weakest precondition rather than the weakest liberal precondition.
  - Generating a variant in addition to an invariant.