Generating Loop Invariants for Program Verification by Transformation

G.W. Hamilton
School of Computing
Dublin City University
Ireland
hamilton@computing.dcu.ie

Loop invariants play a central role in the verification of imperative programs. However, finding these invariants is often a difficult and time-consuming task for the programmer. We have previously shown how program transformation can be used to facilitate the verification of functional programs, but the verification of imperative programs is more challenging due to the need to discover these loop invariants. In this paper, we describe a technique for automatically discovering loop invariants. Our approach is similar to the induction-iteration method, but avoids the potentially exponential blow-up in clauses which can result when using this and other methods. Our approach makes use of the *distillation* program transformation to transform clauses into a simplified form which facilitates the identification of similarities and differences between them and thus discover invariants. We prove that our technique terminates, and demonstrate its successful application to example programs that have proven to be problematic using other approaches.

1 Introduction

The verification of imperative programs generally involves annotating programs with *assertions*, and then using a theorem prover to check these annotations. Central to this annotation process is the use of *loop invariants* which are assertions that are true before and after each iteration of a loop. However, finding these invariants is a difficult and time-consuming task for the programmer, and they are often reluctant to do this. In previous work [15], we have shown how to make use of program transformation in the verification of functional programs. However, the verification of imperative programs is not so straightforward due to the need to discover these invariants prior to verification. In this paper, we describe a technique for automatically discovering loop invariants, thus relieving the programmer of this burden. Our technique relies upon the programmer having provided a *postcondition* for the program; this is much less onerous than providing loop invariants as it generally forms part of the specification of the program.

The technique we describe is similar to the induction-iteration method of Suzuki and Ishihata [31], but we overcome the problems associated with that method which were potential non-termination and exponential blow-up in the size of clauses. Similarly to the induction-iteration method, our technique involves working backward through the iterations of a loop and determining the assertions that are true before each iteration. We use the *distillation* program transformation [14, 16] to transform assertions into a simplified form that facilitates the identification of similarities and differences between them. Commonalities between these assertions are identified, and they are generalised accordingly to give a putative loop invariant that can then be verified. We prove that our technique terminates and demonstrate its successful application to example programs that have proven to be problematic using other approaches.

The remainder of this paper is structured as follows. In Section 2, we describe the simple imperative language that will be used throughout the paper. In Section 3, we provide some background on the use

To appear in EPTCS.

of loop invariants in the verification of programs written in this language. In Section 4, we give a brief overview of the distillation program transformation algorithm which is used to simplify assertions in our approach. In Section 5, we describe our technique for the automatic generation of loop invariants, demonstrate it on some examples, and prove that it terminates. In Section 6, we consider related work and compare these to our own our technique. Section 7 concludes and considers future work.

2 Language

In this section, we introduce our object language, which is a simple imperative programming language.

Definition 2.1 (Language Syntax) The syntax of our object language is as shown in Figure 1.

S	::=	SKIP	Do nothing
	|	$V := E$	Assignment
	|	S_1 ; S_2	Sequence
	|	IF B THEN S_1 ELSE S_2	Conditional
	|	BEGIN VAR $V_1 \ldots V_n$ S END	Local block
	|	WHILE B DO S	While loop

Figure 1: Language Syntax

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

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

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

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

These expressions are defined in a simple functional language with the following syntax.

Definition 2.2 (Expression Syntax) The syntax of expressions in our language is as shown in Figure 2.

E	::=	V	Variable
	|	$C\ E_1 \ldots E_k$	Constructor Application
	|	$\lambda V.E$	λ-Abstraction
	|	F	Function Call
	|	$E_0\ E_1$	Application
	|	CASE E_0 of $P_1 \to E_1 \mid \cdots \mid P_k \to E_k$	Case Expression
	|	E_0 WHERE $F_1 = E_1 \ldots F_n = E_n$	Local Function Definitions
P	::=	$C\ V_1 \ldots V_k$	Pattern

Figure 2: Expression Syntax

An expression can be a variable, constructor application, λ-abstraction, function call, application, CASE or WHERE. Variables introduced by λ-abstractions and CASE patterns are *bound*; all other variables

are *free*. We use $fv(E)$ to denote the free variables of E and write $E \equiv E'$ if E and E' differ only in the names of bound variables.

The constructors are those specified above $(\mathsf{Zero}, \mathsf{Succ}, \mathsf{True}, \mathsf{False})$. We assume a number of pre-defined operators written in this language. For example, for natural number expressions the operators $(+, -, *, /, \%, \hat{})$ implement natural number addition, subtraction, multiplication, division, modulus and exponentiation respectively. For boolean expressions the operators $(\wedge, \vee, \neg, \Rightarrow)$ implement conjunction, disjunction, negation and implication respectively. The relational operators $(<, >, \leq, \geq, =, \neq)$ are also defined.

Definition 2.3 (Substitution) $\theta = \{V_1 \mapsto E_1, \ldots, V_n \mapsto E_n\}$ denotes a *substitution*. If E is an expression, then $E\theta = E\{V_1 \mapsto E_1, \ldots, V_n \mapsto E_n\}$ is the result of simultaneously substituting the expressions $E_1, \ldots, E_n$ for the corresponding variables $V_1, \ldots, V_n$, respectively, in the expression E while ensuring that bound variables are renamed appropriately to avoid name capture.

We reason about the behaviour of our imperative programming language using *Floyd-Hoare style logic* [10, 18]. Specifications in this logic take the form of a triple $\{P\}\ S\ \{Q\}$, where P and Q are boolean expressions that denote the pre- and post-conditions respectively for imperative program S i.e. if P is true, then after execution of S, Q will be true. These are therefore *partial correctness* specifications, and do not say anything about the termination of programs.

Definition 2.4 (Floyd-Hoare Logic) The rules and axioms of Floyd-Hoare logic for our imperative language are as shown in Figure 3.

$$\{P\}\ \mathsf{SKIP}\ \{P\} \qquad \{Q\{V := E\}\}\ V := E\ \{Q\}$$

$$\frac{\{P\}\ S_1\ \{Q\}, \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \qquad \frac{\{P \wedge B\}\ S_1\ \{Q\}, \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \mathsf{IF}\ B\ \mathsf{THEN}\ S_1\ \mathsf{ELSE}\ S_2\ \{Q\}}$$

$$\frac{\{P\}\ S\ \{Q\}, \quad V_1 \ldots V_n \notin fv(P), fv(Q)}{\{P\}\ \mathsf{BEGIN\ VAR}\ V_1 \ldots V_n\ S\ \mathsf{END}\ \{Q\}} \qquad \frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \mathsf{WHILE}\ B\ \mathsf{DO}\ S\ \{I \wedge \neg B\}}$$

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

Figure 3: Floyd-Hoare Logic

In the rule for the while loop, the assertion I is called the *loop invariant*.

3 Loop Invariants

A loop invariant is an assertion that is true before and after each iteration of the loop, and usually needs to be provided by the programmer. A loop which is annotated in this way is denoted by $\mathsf{WHILE}\ B\ \mathsf{DO}\ \{I\}\ S$

Definition 3.1 (Requirements of Loop Invariants) The three requirements of the invariant I of the loop $\{P\}\ \mathsf{WHILE}\ B\ \mathsf{DO}\ \{I\}\ S\ \{Q\}$ are as follows:

1. $P \Rightarrow I$
2. $\{I \wedge B\}\ S\ \{I\}$

3. $(I \wedge \neg B) \Rightarrow Q$

Thus, the precondition P should establish the invariant before executing the loop, the loop body S should maintain the invariant, and the invariant should be sufficient to establish the postcondition Q after exiting the loop.

Example 1 Consider the program shown in Figure 4.

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

Figure 4: Example Program

This program calculates the exponentiation k^n. Say we wish to construct an invariant for the loop in this program which will allow it to be verified. In [11] it is observed that the required invariant is often a weakening of the postcondition for the loop and can be obtained by mutating this postcondition. The assertion $y = k^x$ is an invariant for this loop which is a mutation of the postcondition. However, this invariant is not sufficient to allow verification of this program; the additional invariant $x \leq n$ is also required. In general, the problem of constructing appropriate invariants which are sufficient to allow programs to be verified is undecidable [2]. However, in this paper we show how we can automatically generate invariants which are sufficient to allow a wide range of programs to be verified.

Our approach makes use of the *weakest liberal precondition* originally proposed by Dijkstra [7].

Definition 3.2 (Weakest Liberal Precondition) We define the *weakest liberal precondition* for programs in our language, denoted as $WLP(S,Q)$, where S is a program and Q a postcondition. The result of our calculation is a set of boolean expressions, which are the conjuncts of the weakest liberal precondition. The condition $P = \bigwedge WLP(S,Q)$ if Q is true after execution of S, and no condition weaker than P satisfies this. The key difference of a weakest liberal precondition as opposed to a weakest precondition is that it does not say anything about the termination of programs. The rules for calculating $WLP(S,Q)$ for our programming language are as shown in Figure 5.

$$
\begin{array}{rcl}
WLP(\mathsf{SKIP}, Q) & = & \{Q\} \\
WLP(V := E, Q) & = & \{Q\{V := E\}\} \\
WLP(S_1; S_2, Q) & = & \bigcup\{WLP(S_1, P) | P \in WLP(S_2, Q)\} \\
WLP(\mathsf{IF}\ B\ \mathsf{THEN}\ S_1\ \mathsf{ELSE}\ S_2, Q) & = & \{B \Rightarrow P | P \in WLP(S_1, Q)\} \cup \{\neg B \Rightarrow P | P \in WLP(S_2, Q)\} \\
WLP(\mathsf{WHILE}\ B\ \mathsf{DO}\ \{I\}\ S, Q) & = & \{I\} \cup \{(B \wedge I) \Rightarrow P | P \in WLP(S, I)\} \cup \{(\neg B \wedge I) \Rightarrow Q\}
\end{array}
$$

Figure 5: Weakest Liberal Precondition

Note that the weakest liberal precondition calculation for a loop requires that it has already been annotated with its invariant. This implies that we should apply our techniques to inner loops first to determine their invariant before applying them to outer loops.

4 Distillation

The predicates produced in our approach are simplified using the *distillation* transformation [14, 16]. Distillation is a fold/unfold program transformation that builds on top of positive supercompilation [30], but is more powerful, thus allowing more simplifications to be performed. The main distinguishing characteristic between the two algorithms is that in distillation, generalisation and folding are performed with respect to recursive terms, while in positive supercompilation they are not. In the work described here, we transform predicates into a simplified form that facilitates the identification of similarities and differences between them. In particular, built-in associative operators (such as +,*,$\wedge$,$\vee$) are always transformed into *right-associative* form.

4.1 Embedding

Generalisation is performed if the predicate obtained from distillation is an *embedding* of a previously distilled one. The form of embedding which we use to inform this process is known as *homeomorphic embedding*. The homeomorphic embedding relation was derived from results by Higman [17] and Kruskal [22] and was defined within term rewriting systems [6] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [29], distillation [14, 16], partial evaluation [24] and partial deduction [3, 23].

Definition 4.1 (Expression Embedding) An expression E is *embedded* in expression E' if $E \trianglelefteq E'$, where the binary relation $\trianglelefteq$ is defined as follows.

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

The first rule here is for variables, the second is a *diving* rule and the third is a *coupling* rule. Diving detects a sub-expression embedded in a larger expression, and coupling matches all the sub-expressions of two expressions which have the same top-level constructor. Bound variables are handled by this relation by requiring that they have the same de Bruijn indices. We write $E \preceq E'$ if expression E is coupled with expression E'.

4.2 Generalisation

Definition 4.2 (Generalisation of Expressions) The generalisation of expression E with respect to expression E' (denoted by $E \sqcap E'$) is defined as shown below.

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

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 $E''\theta' \equiv E'$. Within these rules, if both expressions have the same constructor at the outermost level, this is made the outermost constructor of the resulting generalised expression, and the corresponding sub-expressions within the constructor applications are then generalised. Otherwise, both expressions are replaced by the same variable.

Definition 4.3 (Most Specific Generalisation) A 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, denoted by $E \triangle E'$, of expressions E and E' is computed by exhaustively applying the following rewrite rule to the triple obtained from the generalisation $E \sqcap E'$:

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

This minimises the substitutions by identifying common substitutions which were previously given different names.

5 Automatic Generation of Loop Invariants

In order to calculate loop invariants, starting from the postcondition, we work our way backwards through each iteration of the loop generating successive approximations to the loop invariant. 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. If there are a number of different possible paths through the loop, then a number of possible preconditions will be calculated for it; these are collapsed into a single precondition by being generalised with respect to each other, thus producing a single approximation for each loop iteration.

Our algorithm for the automatic generation of an invariant for the loop WHILE B DO S with postcondition Q is as shown in Figure 6.

$f\ (distill(\neg B \wedge Q))\ \emptyset$
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 $f\ (\bigwedge\{distill(B \wedge P') \mid P' \in WLP(S,P)\})\ (\phi \cup \{P\})$

Figure 6: Algorithm for Finding Loop Invariants

Here, P is the current predicate (initially equivalent to $\neg B \wedge Q$) and ϕ is the set of previous approximations to the invariant (initially empty). If P is a renaming of a predicate in ϕ, then P is returned as the putative invariant. If there is a predicate Q in ϕ which is an embedding of P, then P is generalised with respect to Q, and the algorithm is further applied to this generalisation. Otherwise, the set of new possible predicates is calculated using $WLP(S,P)$, simplified using distillation, and then generalised together. The algorithm is then further applied to the resulting generalised predicate with P added to ϕ.

The generated invariant may contain generalisation variables; inductive definitions for these variables can be determined using the three requirements for loop invariants (Definition 3.1). We try to find values for these variables that satisfy each of these requirements using our Poitín theorem prover [13][1]. If we are not able to satisfy all three of these requirements, then we have failed in finding a suitable invariant.

[1] This could also be done using a SAT solver.

For the loop $\{P\}$ 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:

$$\{I\{v := v_i\} \wedge B\}\ S\ \{I\{v := v_{i+1}\}\}$$

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 1 We illustrate this algorithm by applying it to the example program in Figure 4.

Firstly, we calculate the logical assertion which is true if the loop is exited:

$$\neg(\mathsf{x} < \mathsf{n}) \wedge \mathsf{y} = \mathsf{k}^{\wedge}\mathsf{n} \tag{1}$$

This is simplified by distillation to the following[2]:

$$\mathsf{x} \geq \mathsf{n} \wedge \mathsf{y} = \mathsf{k}^{\wedge}\mathsf{n} \tag{2}$$

Then, we calculate the set of logical assertions which can be true before the final execution of the loop body:

$$\{\mathsf{x} < \mathsf{n} \wedge \mathsf{P} \mid \mathsf{P} \in \mathsf{WLP}(\mathsf{BEGIN}\ \mathsf{x} := \mathsf{x}+1; \mathsf{y} := \mathsf{y} * \mathsf{k}\ \mathsf{END}, \mathsf{x} \geq \mathsf{n} \wedge \mathsf{y} = \mathsf{k}^{\wedge}\mathsf{n})\} \tag{3}$$

This gives the following predicate:

$$\mathsf{x} < \mathsf{n} \wedge \mathsf{x}+1 \geq \mathsf{n} \wedge \mathsf{y} * \mathsf{k} = \mathsf{k}^{\wedge}\mathsf{n} \tag{4}$$

which is simplified to the following by distillation:

$$\mathsf{x}+1 = \mathsf{n} \wedge \mathsf{y} * \mathsf{k} = \mathsf{k}^{\wedge}\mathsf{n} \tag{5}$$

This is not an embedding of (2), so the calculation continues. We next calculate the set of logical assertions which can be true before the penultimate execution of the loop body:

$$\{\mathsf{x} < \mathsf{n} \wedge \mathsf{P} \mid \mathsf{P} \in \mathsf{WLP}(\mathsf{BEGIN}\ \mathsf{x} := \mathsf{x}+1; \mathsf{y} := \mathsf{y} * \mathsf{k}\ \mathsf{END}, \mathsf{x}+1 = \mathsf{n} \wedge \mathsf{y} * \mathsf{k} = \mathsf{k}^{\wedge}\mathsf{n})\} \tag{6}$$

This gives the following predicate:

$$\mathsf{x} < \mathsf{n} \wedge (\mathsf{x}+1)+1 = \mathsf{n} \wedge (\mathsf{y} * \mathsf{k}) * \mathsf{k} = \mathsf{k}^{\wedge}\mathsf{n} \tag{7}$$

which is simplified to the following by distillation:

$$\mathsf{x}+2 = \mathsf{n} \wedge \mathsf{y} * (\mathsf{k} * \mathsf{k}) = \mathsf{k}^{\wedge}\mathsf{n} \tag{8}$$

[2]For this and all following examples, the result of distillation is simplified by folding any instances of the definitions of the pre-defined operators of our language; the results would be too unwieldy otherwise.

We can see that (8) is an embedding of (5) (2 is $\mathsf{Succ}(\mathsf{Succ}(\mathsf{Zero}))$ and is an embedding of 1 which is $\mathsf{Succ}(\mathsf{Zero})$), so (8) is generalised to produce the following:

$$\mathsf{x}+\mathsf{v}=\mathsf{n}\wedge\mathsf{y}*\mathsf{w}=\mathsf{k}^{\wedge}\mathsf{n} \tag{9}$$

where v and w are new generalisation variables. This is not an embedding of (5) or (2), so the set of logical assertions which are true before execution of the loop body are now re-calculated as follows:

$$\{\mathsf{x}<\mathsf{n}\wedge\mathsf{P}\mid\mathsf{P}\in\mathsf{WLP}(\mathsf{BEGIN}\,\mathsf{x}:=\mathsf{x}+1;\mathsf{y}:=\mathsf{y}*\mathsf{k}\,\mathsf{END},\mathsf{x}+\mathsf{v}=\mathsf{n}\wedge\mathsf{y}*\mathsf{w}=\mathsf{k}^{\wedge}\mathsf{n})\} \tag{10}$$

This gives the following predicate:

$$\mathsf{x}<\mathsf{n}\wedge(\mathsf{x}+1)+\mathsf{v}=\mathsf{n}\wedge(\mathsf{y}*\mathsf{k})*\mathsf{w}=\mathsf{k}^{\wedge}\mathsf{n} \tag{11}$$

which is simplified to the following by distillation:

$$\mathsf{x}+(\mathsf{v}+1)=\mathsf{n}\wedge\mathsf{y}*(\mathsf{k}*\mathsf{w})=\mathsf{k}^{\wedge}\mathsf{n} \tag{12}$$

We can see that (12) is an embedding of (9), so (12) is generalised to produce the following:

$$\mathsf{x}+\mathsf{v}'=\mathsf{n}\wedge\mathsf{y}*\mathsf{w}'=\mathsf{k}^{\wedge}\mathsf{n} \tag{13}$$

where v' and w' are new generalisation variables. We can now see that (13) is a renaming of (9), so (13) is our putative invariant.

We now try to find inductive definitions for the generalisation variables v' and w' from the three requirements of loop invariants given in Definition 3.1, which we do using our theorem prover Poitín.

The initial values of the generalisation variables, given by v'_0 and w'_0, can be determined using the first invariant requirement as follows:

$$\mathsf{n}\geq 0\wedge\mathsf{x}=0\wedge\mathsf{y}=1\Rightarrow\mathsf{x}+\mathsf{v}'_0=\mathsf{n}\wedge\mathsf{y}*\mathsf{w}'_0=\mathsf{k}^{\wedge}\mathsf{n} \tag{14}$$

The assignments $v'_0 := n$ and $w'_0 := k^{\wedge}n$ satisfy this assertion.

The inductive values of the generalisation variables, given by v'_{i+1} and w'_{i+1}, can be determined using the second invariant requirement as follows:

$$\mathsf{x}+\mathsf{v}'_{\mathsf{i}}=\mathsf{n}\wedge\mathsf{y}*\mathsf{w}'_{\mathsf{i}}=\mathsf{k}^{\wedge}\mathsf{n}\wedge\mathsf{x}<\mathsf{n}\Rightarrow(\mathsf{x}+1)+\mathsf{v}'_{\mathsf{i}+1}=\mathsf{n}\wedge(\mathsf{y}*\mathsf{k})*\mathsf{w}'_{\mathsf{i}+1}=\mathsf{k}^{\wedge}\mathsf{n} \tag{15}$$

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

The final values of the generalisation variables, given by v'_n and w'_n, can be determined using the third invariant requirement as follows:

$$\mathsf{x}+\mathsf{v}_{\mathsf{n}}=\mathsf{n}\wedge\mathsf{y}*\mathsf{w}_{\mathsf{n}}=\mathsf{k}^{\wedge}\mathsf{n}\wedge\neg(\mathsf{x}<\mathsf{n})\Rightarrow\mathsf{y}=\mathsf{k}^{\wedge}\mathsf{n} \tag{16}$$

The assignments $v_n := 0$ and $w_n = 1$ satisfy this assertion.

The discovered invariant is therefore equivalent to the following:

$$\mathsf{x}\leq\mathsf{n}\wedge\mathsf{y}=\mathsf{k}^{\wedge}\mathsf{x} \tag{17}$$

Example 2 Consider the following example 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}
```

This program also calculates the exponentiation $k^{\wedge}n$. This example is problematic using other approaches because of the presence of the conditional inside the loop, which causes an exponential blow-up in the size of the generated predicates; we show how this blow-up is avoided using our approach. In the following, we use S to denote the body of the loop in the above program. Firstly, we calculate the logical assertion which is true if the loop is exited:

$$\neg(x > 0) \wedge y = k^{\wedge}n \tag{1}$$

This is simplified by distillation to the following:

$$x \leq 0 \wedge y = k^{\wedge}n \tag{2}$$

Then, we calculate the set of logical assertions which can be true before the final execution of the loop body:

$$\{x > 0 \wedge P \mid P \in \mathsf{WLP}(S, x \leq 0 \wedge y = k^{\wedge}n)\} \tag{3}$$

This gives the following predicates:

$$x > 0 \wedge x\%2 = 1 \Rightarrow x/2 \leq 0 \wedge y * z = k^{\wedge}n \tag{4}$$

$$x > 0 \wedge \neg(x\%2 = 1) \Rightarrow x/2 \leq 0 \wedge y = k^{\wedge}n \tag{5}$$

(5) is found to be false as a result of distillation, but (4) is simplified to the following:

$$x = 1 \wedge y * z = k^{\wedge}n \tag{6}$$

This is not an embedding of (2), so the calculation continues. We next calculate the set of logical assertions which can be true before the penultimate execution of the loop body:

$$\{x > 0 \wedge P \mid P \in \mathsf{WLP}(S, x = 1 \wedge y * z = k^{\wedge}n)\} \tag{7}$$

This gives the following predicates:

$$x > 0 \wedge x\%2 = 1 \Rightarrow x/2 = 1 \wedge (y * z) * (z * z) = k^{\wedge}n \tag{8}$$

$$x > 0 \wedge \neg(x\%2 = 1) \Rightarrow x/2 = 1 \wedge y * (z * z) = k^{\wedge}n \tag{9}$$

which are simplified to the following by distillation:

$$x = 3 \wedge y * (z * (z * z)) = k^{\wedge} n \tag{10}$$
$$x = 2 \wedge y * (z * z) = k^{\wedge} n \tag{11}$$

These are generalised with respect to each other to give the following:

$$x = v \wedge y * (z * w) = k^{\wedge} n \tag{12}$$

where v and w are new generalisation variables. This is not an embedding of (6) or (2), so the set of logical assertions which are true before execution of the loop body are now re-calculated as follows:

$$\{x > 0 \wedge P \mid P \in \mathsf{WLP}(S, x = v \wedge y * (z * w) = k^{\wedge} n)\} \tag{13}$$

This gives the following predicates:

$$x > 0 \wedge x\%2 = 1 \Rightarrow x/2 = v \wedge (y * z) * ((z * z) * w) = k^{\wedge} n \tag{14}$$
$$x > 0 \wedge \neg(x\%2 = 1) \Rightarrow x/2 = v \wedge y * ((z * z) * w) = k^{\wedge} n \tag{15}$$

which are simplified to the following by distillation:

$$x = v * 2 + 1 \wedge y * (z * (z * (z * w))) = k^{\wedge} n \tag{16}$$
$$x = v * 2 \wedge y * (z * (z * w)) = k^{\wedge} n \tag{17}$$

These are generalised with respect to each other to give the following:

$$x = v' \wedge y * (z * (z * w')) = k^{\wedge} n \tag{18}$$

where v' and w' are new generalisation variables. We can see that (18) is an embedding of (12), so (18) is generalised to give the following:

$$x = v'' \wedge y * (z * w'') = k^{\wedge} n \tag{19}$$

where v'' and w'' are new generalisation variables. We can now see that (19) is a renaming of (12), so (19) is our putative invariant.

We now try to find inductive definitions for the generalisation variables v'' and w'' from the three requirements of loop invariants given in Definition 3.1, which we do using our theorem prover Poitín.

The initial values of the generalisation variables, given by v''_0 and w''_0, can be determined using the first invariant requirement as follows:

$$n \geq 0 \wedge x = n \wedge y = 1 \wedge z = k \Rightarrow x = v''_0 \wedge y * (z * w''_0) = k^{\wedge} n \tag{20}$$

The assignments $v''_0 := n$ and $w''_0 := k^{\wedge}(n-1)$ satisfy this assertion.

The inductive values of the generalisation variables, given by v''_{i+1} and w''_{i+1}, can be determined using the second invariant requirement as follows:

$$x = v''_i \wedge y * (z * w''_i) = k^{\wedge} n \wedge x > 0 \wedge x\%2 = 0 \Rightarrow x/2 = v''_{i+1} \wedge (y * z) * ((z * z) * w''_{i+1}) = k^{\wedge} n \tag{21}$$
$$x = v''_i \wedge y * (z * w''_i) = k^{\wedge} n \wedge x > 0 \wedge \neg(x\%2 = 1) \Rightarrow x/2 = v''_{i+1} \wedge y * ((z * z) * w''_{i+1}) = k^{\wedge} n \tag{22}$$

The assignments $v''_{i+1} := v''_i/2$ and $w''_{i+1} := w''_i/(z*z)$ satisfy the first assertion and the assignments $v''_{i+1} := v''_i/2$ and $w''_{i+1} := w''_i/z$ satisfy the second assertion.

The final values of the generalisation variables, given by v_n and w_n, can be determined using the third invariant requirement as follows:

$$\mathsf{x} = \mathsf{v''_n} \wedge \mathsf{y} * (\mathsf{z} * \mathsf{w''_n}) = \mathsf{k}^{\wedge}\mathsf{n} \wedge \neg(\mathsf{x} > 0) \Rightarrow \mathsf{y} = \mathsf{k}^{\wedge}\mathsf{n} \tag{23}$$

The assignments $v''_n := 0$ and $w''_n = 1/z$ satisfy this assertion.

The discovered invariant is therefore equivalent to the following:

$$\mathsf{x} = \mathsf{n}/2^{\mathsf{j}} \wedge \mathsf{y} = \mathsf{k}^{\wedge}(\mathsf{n}\%2^{\mathsf{j}}) \tag{24}$$

5.1 Termination

In order to prove that our loop invariant algorithm always terminates, we firstly need to show that in any infinite sequence of predicates $P_0, P_1, \ldots$ there definitely exists some $i < j$ where $P_i \preceq P_j$. This amounts to proving that the embedding relation $\preceq$ is a *well-quasi order*.

Definition 5.1 (Well-Quasi Order) A well-quasi order on a set S is a reflexive, transitive relation $\leq$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \leq s_j$.

Lemma 5.2 ($\preceq$ is a Well-Quasi Order) The embedding relation $\preceq$ is a *well-quasi order* on any sequence of predicates.

Proof. The proof is similar to that given in [21]. It involves showing that there are a finite number of functors (function names and constructors) in the language. Applications of different arities are replaced with separate constructors; we prove that arities are bounded so there are a finite number of these. We also replace case expressions with constructors. Since bound variables are defined using de Bruijn indices, each of these are replaced with separate constructors; we also prove that de Bruijn indices are bounded. The overall number of functors is therefore finite, so Kruskal's tree theorem can then be applied to show that $\preceq$ is a well-quasi-order. □

Theorem 5.3 (Termination of the Loop Invariant Algorithm) The loop invariant algorithm always terminates.

Proof. The proof is by contradiction. If the loop invariant algorithm did not terminate then the set of invariants generated as successive approximations to the invariant must be infinite. Every new predicate which is added to the set of approximations cannot have any of the previously generated predicates on this set embedded within it by the homeomorphic embedding relation $\preceq$, since either generalisation would have been performed or a renaming encountered and the algorithm terminated. However, this contradicts the fact that $\preceq$ is a well-quasi-order (Lemma 5.2). □

6 Related Work

The main approaches to the automatic generation of loop invariants include abstract interpretation, proof planning, dynamic methods, using heuristics and the induction-iteration method. The earliest methods for the automatic generation of loop invariants involved static analysis. *Abstract interpretation* is a symbolic execution of programs over abstract domains (such as predicate abstraction domains or polyhedral abstraction domains) that over-approximates the semantics of loop iteration. *Predicate abstraction* domains [1, 12, 27, 5, 9] replace predicates with variables, which is similar to the generalisation we perform in our approach. Constraint-based techniques rely on sophisticated decision procedures over non-trivial mathematical domains (such as polynomials [28] or convex polyhedra [4]) to represent concisely the semantics of loops with respect to certain properties. Loop invariants in these forms are extremely useful but rarely sufficient to prove full functional correctness of programs.

In [9], Flanagan and Qadeer describe the use of predicate abstraction to generate loop invariants. Their approach differs from our own in that predicates are obtained by working forwards from the precondition through successive iterations of the loop, as opposed to backwards from the postcondition in our approach. A strongest postcondition semantics is therefore used in [9] as opposed to our weakest precondition approach. Loop invariants are computed by iterative approximation. The first approximation is obtained by abstracting the set of reachable states at loop entry. Each successive approximation enlarges the current approximation to include the states reachable by executing the loop body once from the states in the current approximation. The iteration terminates in a loop invariant since the abstract domain is finite. However, this approach does suffer from the drawback that the approximations can grow exponentially as they are a disjunction of the approximations for all the reachable states. This exponential growth is avoided in our approach. Also, we argue that working forwards from the precondition makes it harder to find the required invariant since (as observed in [11]), the required invariant is often a weakening of the postcondition.

In [20], a *proof planning* approach is used to synthesise loop invariants. This approach makes use of failed attempts to prove a putative invariant correct. The proof attempts are applied to the verification conditions generated for the putative invariant. If these proof attempts fail, the failure is analysed using *proof critics*. One such critic is the generalisation critic, which performs generalisation in a similar way to that described in our work, and is used to update the putative invariant to one which is more likely to be correct. One drawback of this approach is that the original putative invariant has to be guessed, although the postcondition is a good first guess. Another drawback is knowing which critics to apply when, since multiple critics can discover the invariant, but some may do so more efficiently than others. Also, it is not clear how this method could be applied to nested loops.

In [8], invariants are discovered dynamically. Using this approach, the program is run over a test suite of inputs. The corresponding outputs are analysed for patterns and relationships among the variables. Candidate invariants are guessed by trying out a pre-defined set of user-provided templates (including comparisons between variables, simple inequalities, and simple list comprehensions). These candidate invariants are then tested against several program runs; the invariants that are not violated in any of the runs are retained as likely invariants. This inference is not sound and only gives an educated guess. However, a prototype tool called Daikon was implemented using these techniques, and has worked well in practice and many of the guessed invariants are sound.

In [11], Furia and Meyer describe the use of *heuristics* to synthesise loop invariants. This work is based on the observation that the required invariant is often a weakening of the postcondition for the loop and can be obtained by mutating this postcondition. The core idea is to generate candidate invariants by mutating postconditions according to a few commonly recurring patterns. Although this idea works well

in many cases, it is not capable of generating the required invariant for the example program in Figure 4, as this requires the addition of an extra clause to the postcondition ($y \leq n$), which is not one of described heuristics.

The previous work which is closest to our own is the *induction-iteration* method of Suzuki and Ishihata [31]. This method works as follows for the program $\{P\}\ S_1; \mathsf{WHILE}\ B\ \mathsf{DO}\ S_2\ \{Q\}$ with precondition P and postcondition Q. Firstly, the logical assertion which is true if the loop is exited is calculated in a similar way to our technique:

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

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

$$P_{i+1} = (B \Rightarrow WLP(S_2, P_i)$$

The weakest liberal precondition of the loop is given by $\bigwedge_{i=0}^{\infty} P_i$. In order to calculate this finitely, a number of successive approximations are calculated for it until one is found that is a loop invariant, where the j^{th} approximation is given by $I_j = \bigwedge_{i=0}^{j} P_i$. It then has to be shown that this approximation is true on entry to the loop and is also a loop invariant:

$$P \Rightarrow WLP(S_1, I_j) \tag{1}$$

$$(I_j \wedge B) \Rightarrow WLP(S_2, I_j) \tag{2}$$

(2) is equivalent to the following:

$$I_j \Rightarrow P_{j+1} \tag{3}$$

This therefore suggests an iterative approach to finding the loop invariant. Successive values for I_j can be computed making use of the previous values. If (3) is satisfied for the current value of I_j, then we are done and I_j is the required invariant. If (1) is not satisfied for the current value of I_j, then we have failed to find a suitable invariant. Otherwise, we carry on the iteration to I_{j+1}.

One problem with this approach is that it is not guaranteed to terminate. This is avoided by limiting the number of iterations. It is found that in practice, for most of the small examples tried, very few iterations are actually required. Our approach is always guaranteed to terminate, but of course it may not be able to find a suitable invariant. Another problem with this approach is that there can be an exponential blow-up in clauses into increasingly larger conjunctions. This is particularly the case for conditionals, and can degrade I_j to such an extent that it never converges to a loop invariant. This problem is avoided in [31] by cleverly designing the theorem prover to avoid this potential exponential blow-up in clauses. In our approach, this problem is avoided by combining the conjuncts using generalisation. Finally, the seminal work in [31] does not show how to deal with nested loops. However, an extension to the technique which does this is described by Xu et al. [32]. A similar approach can also be used for our proposed technique to deal with nested loops.

7 Conclusions and Further Work

In this paper we have described a technique for automatically discovering loop invariants. The technique we describe is similar to the induction-iteration method of Suzuki and Ishihata [31], but we overcome the problems associated with that method. One of these problems was the potential non-termination

of the induction-iteration method; our technique is guaranteed to terminate. Another problem with the induction-iteration method is the potential exponential blow-up in clauses into increasingly larger conjunctions. Our technique avoids this through the combination of these conjuncts using generalisation. We have successfully demonstrated our technique on example imperative programs that have proven to be problematic using other approaches.

There are a number of possible directions for further work. Firstly, we need to extend our techniques to languages with richer features. For example, we could extend the language to manipulate unbounded data structures such as arrays. For such constructs, the required loop invariants need to be universally quantified, but this can be handled by our theorem prover Poitín, so should not present a problem for our technique. Another way in which the language could be extended would be to handle pointers. Separation logic [25, 26] extends Floyd-Hoare logic to be able to handle pointers, so this seems to be an obvious basis for the extension of our technique. It has already been shown by Ireland [19] how his approach to invariant generation can be extended to handle pointers by making use of separation logic.

One other possible direction for further work is extending our technique to deal with the termination of programs. This would involve calculating the *weakest precondition* rather than the weakest liberal precondition as we do here. This would require the generation of a *variant* in addition to an invariant, and the refinement of the invariant to show that the variant is decreased on each iteration of the loop. This appears to be a lot more challenging than the problem which is tackled here.

References

[1] T. Agerwala & J. Misra (1978): *Assertion Graphs for Verifying and Synthesizing Programs*. Technical Report 83, University of Texas, Austin.

[2] A. Blass & Y. Gurevich (2001): *Inadequacy of Computable Loop Invariants*. *ACM Transactions on Computational Logic* 2(1), pp. 1–11.

[3] R. Bol (1993): *Loop Checking in Partial Deduction*. *Journal of Logic Programming* 16(1–2), pp. 25–46.

[4] P. Cousot & N. Halbwachs (1978): *Automatic Discovery of Linear Restraints Among Variables of a Program*. In: *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 84–96.

[5] S. Das, D.L. Dill & S. Park (1999): *Experience With Predicate Abstraction*. In: *International Conference on Computer Aided Verification*, *Lecture Notes in Computer Science* 1633, Springer, pp. 160–171.

[6] N. Dershowitz & J.-P. Jouannaud (1990): *Rewrite Systems*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science*, Elsevier, pp. 244–320.

[7] E.W. Dijkstra (1975): *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. *Communications of the ACM* 18, pp. 453–457.

[8] M. Ernst, J. Cockrell, W. Griswold & D. Notkin (2001): *Dynamically Discovering Likely Program Invariants to Support Program Evolution*. *IEEE Transactions in Software Engineering* 27(2), pp. 1–25.

[9] C. Flanagan & S. Qadeer (2002): *Predicate Abstraction for Software Verification*. In: *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, pp. 191–202.

[10] R.W. Floyd (1967): *Assigning Meanings to Programs*. In: *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19, pp. 19–32.

[11] C.A. Furia & B. Meyer (2010): *Inferring Loop Invariants Using Postconditions*. In: *Fields of Logic and Computation*, Springer, pp. 277–300.

[12] S. Graf & H. Saidi (1997): *Construction of Abstract State Graphs With PVS*. In: *International Conference on Computer Aided Verification*, *Lecture Notes in Computer Science* 1254, Springer, pp. 72–83.

[13] G. W. Hamilton (2006): *Poitín: Distilling Theorems From Conjectures*. Electronic Notes in Theoretical Computer Science 151(1), pp. 143–160.

[14] G.W. Hamilton (2007): *Distillation: Extracting the Essence of Programs*. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, pp. 61–70.

[15] G.W. Hamilton (2007): *Distilling Programs for Verification*. Electronic Notes in Theoretical Computer Science 190(4), pp. 17–32.

[16] G.W. Hamilton & N.D. Jones (2012): *Distillation With Labelled Transition Systems*. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM, pp. 15–24.

[17] G. Higman (1952): *Ordering by Divisibility in Abstract Algebras*. Proceedings of the London Mathematical Society 2, pp. 326–336.

[18] C.A.R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. Communications of the ACM 12, pp. 576–583.

[19] A. Ireland (2006): *Towards Automatic Assertion Refinement for Separation Logic*. In: Proceedings of the International Conference on Automated Software Engineering, pp. 209–312.

[20] A. Ireland & J. Stark (1997): *On the Automatic Discovery of Loop Invariants*. In: Fourth Nasa Langley Formal Methods Workshop, pp. 137–152.

[21] I. Klyuchnikov (2010): *Supercompiler HOSC 1.1: Proof of Termination*. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow.

[22] J.B. Kruskal (1960): *Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture*. Transactions of the American Mathematical Society 95, pp. 210–225.

[23] M. Leuschel (1998): *On the Power of Homeomorphic Embedding for Online Termination*. In: Proceedings of the International Static Analysis Symposium, Pisa, Italy, pp. 230–245.

[24] R. Marlet (1994): *Vers une Formalisation de l'Évaluation Partielle*. Ph.D. thesis, Université de Nice - Sophia Antipolis.

[25] P. O'Hearn, J. Reynolds & Y. Hongseok (2001): *Local Reasoning About Programs That Alter Data Structures*. In: Proceedings of Computer Science Logic, Lecture Notes in Computer Science 2142, pp. 1–19.

[26] J.C. Reynolds (2002): *Separation Logic: A Logic for Shared Mutable Data Structures*. In: Proceedings of the Symposium on Logic in Computer Science, pp. 55–74.

[27] H. Saidi & N. Shankar (1999): *Abstract and Model Check While You Prove*. In: International Conference on Computer Aided Verification, Lecture Notes in Computer Science 1633, Springer, pp. 443–454.

[28] S. Sankaranarayanan, H. Sipma & Z. Manna (2004): *Non-Linear Loop Invariant Generation Using Gr obner Bases*. In: Proceedings of the ACM Symposium on Principles of Programming Languages, pp. 318–329.

[29] M.H. Sørensen & R. Glück (1994): *An Algorithm of Generalization in Positive Supercompilation*. Lecture Notes in Computer Science 787, pp. 335–351.

[30] M.H. Sørensen, R. Glück & N.D. Jones (1996): *A Positive Supercompiler*. Journal of Functional Programming 6(6), pp. 811–838.

[31] N. Suzuki & K. Ishihata (1977): *Implementation of an Array Bound Checker*. In: 4th ACM Symposium on Principles of Programming Languages, ACM Press, pp. 132–143.

[32] Z. Xu & B. Reps, T. amd Miller (2000): *Safety Checking of Machine Code*. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, pp. 70–82.