Generalisation of Induction Formulae based on Proving by Symbolic Execution

Angela Wallenburg*

Chalmers University of Technology and University of Gothenburg, Sweden
angelaw@chalmers.se

Abstract

Induction is a powerful method that can be used to prove the total correctness of program loops. Unfortunately the induction proving process in an interactive theorem prover is often very cumbersome. In particular it can be difficult to find the right induction formula. We describe a method for generalising induction formulae by analysing a symbolic proof attempt in a semi-interactive first-order theorem prover. Based on the proof attempt we introduce universally quantified variables, meta-variables and sets of constraints on these. The constraints describe the conditions for a successful proof. By the help of examples, we outline some classes of problems and their associated constraint solutions, and possible ways to automate the constraint solving.

1 Introduction

Induction is a powerful proof method that can be used to prove that a property holds for a possibly infinite sequence of elements. In many program verification applications this is an essential tool. An excellent example is in proving the total correctness of loops, which is the focus of this paper. A standard induction proof requires proving that a property, the *induction formula*, holds for the first element in a sequence and then proving that if it holds for an arbitrary element in the sequence, then it holds for the next one. Though this basic idea of induction may seem straightforward, in a practical program verification setting induction is notoriously difficult to use and to automate.

A central complication is the degree of incompleteness that occurs in practice; that is, for a given property it is common that an inductive proof attempt fails even if the property is valid. To overcome this problem we can prove a property that is stronger than the one originally desired and then prove that the stronger property implies the original proof obligation, i.e. that it is a *generalisation*. We use a rule that we call natInduct both to prove that the (stronger) induction formula ϕ is valid *and* to prove that if ϕ holds, then the original proof obligation Δ holds:

$$\textsf{natInduct}\ \frac{\Gamma \Longrightarrow \phi(0),\ \Delta \quad \Gamma \Longrightarrow \forall n.\ 0 \leq n \wedge \phi(n) \rightarrow \phi(n+1),\ \Delta \quad \Gamma,\ \forall n.\ 0 \leq n \wedge \phi(n) \Longrightarrow \Delta}{\Gamma \Longrightarrow \Delta}$$

In the calculus that we use $\alpha, \ldots, \gamma \Rightarrow \delta, \ldots, \varphi$ has the semantics $\alpha \wedge \ldots \wedge \gamma \rightarrow \delta \vee \ldots \vee \varphi$. The starting assumption is Γ and $\Longrightarrow$ is called the sequent arrow. For an early reference to sequent calculus, see [10], and for the sequent calculus that is the foundation of our system, see [11]. Note the presence of the induction formula, ϕ, in all three branches of an inductive proof: 1) in the *base case*: $\phi(0)$, 2) in the *step case*: $\forall n.0 \leq n \wedge \phi(n) \rightarrow \phi(n+1)$, and 3) in the *use case*: $\forall n.0 \leq n \wedge \phi(n) \rightarrow \Delta$. Any change to the induction formula propagates to all

*Now at Praxis High Integrity Systems Ltd, Bath, U.K.

branches of the proof. In particular a generalisation of the induction formula both introduces more assumptions and—simultaneously and in different branches of the tree—increases the proof burden. Therefore a patch to the induction formula based on the information from one failed proof branch is likely to cause another branch to fail. A further problem—particularly for a real programming language with exceptions, aliasing etc.—is that the reasoning in all the "straight stretches" in between the induction-specific points of the proof is also non-trivial and has to be repeated for every failed proof attempt, which in an interactive verification system usually involves manual steps.

This paper presents an idea for finding the right induction formula by generalising the original proof obligation based on information from failed proof attempts. First we analyse the state changes during symbolic execution of the loop body to add suitable universally quantified variables and placeholders for parts of the induction formula, typically the postcondition. Next we make a proof attempt with the modified formula, the induction formula guess. Our induction proving process is syntactically driven and consists of a number of well-defined steps, all of which are necessary for a successful proof. Every open proof goal in the resulting proof attempt therefore generates a constraint on the values of the placeholders introduced in our induction formula guess. We know that if the constraints can be solved, then the proof will succeed. While solving the generated second-order constraints is undecidable in the general case, there are a number of classes of problems where the solutions are simple enough to be candidates for automatic constraint solving techniques.

Section 2 describes the verification system properties that our method builds on. Section 3 gives a motivating example, which illustrates the induction proving process, and the proposed constraint generation method. Section 4 describes the constraint generation method on a general level. Sections 5 and 6 give two additional examples of the constraint generation. While solving the generated constraints is beyond the scope of this article, Section 7 gives an overview of how the constraints can be solved by hand, and of some classes of possible automatic solutions. We then describe some limitations of the method and possible future work, some related work on induction formula generation and loop invariant generation, and finally provide a summary and some conclusions.

2 Verification Setting

In this section we describe some important properties of the verification system that our method is developed for. A typical formula in this system is written as:

$$\forall\ vars.(precondition \rightarrow \{state\ description\}\ \langle \texttt{program} \rangle\ postcondition)$$

and it means that if the precondition holds, and if the program is executed from the starting state described within the curly brackets, then it terminates in a state where the postcondition holds.

The calculus that we use can be described as a variant of weakest precondition calculus. The underlying logic is a version of dynamic logic (DL) [14]. It differs from the original DL in that there is a distinction between logical variables and program variables. Only logical variables can be quantified over, for example *vars* in the typical formula above have to be logical variables.

Furthermore, the calculus calculates the substitutions with respect to an *arbitrary* postcondition. Our analysis relies on this calculation of general substitutions, which we call *updates*. For example, if we would apply the assignment rule(s) to the formula $\{\texttt{i} := 3\}\langle\texttt{j = 2; i--;}\rangle\texttt{i} \doteq \texttt{j}$, we would get $\{\texttt{i} := 3 \,||\, \texttt{j} := 2\}\langle\texttt{i--;}\rangle\texttt{i} \doteq \texttt{j}$, and then after another assignment rule application

```
while (0 < i) {
  i--;
  s = s + j;
}
```

Figure 1: A program which calculates the product of two numbers. The same example appears in [26].

$$
\begin{array}{l}
\Longrightarrow \\
\forall il.\forall jl.(0 \leq il \rightarrow \\
\{\mathtt{i} := il \,||\, \mathtt{j} := jl \,||\, \mathtt{s} := 0\} \\
\langle\, \texttt{while (0 < i) \{} \\
\quad \texttt{i--;} \\
\quad \texttt{s = s + j;} \\
\quad \texttt{\}} \,\rangle\, \mathtt{s} \doteq il * jl)
\end{array}
$$

Figure 2: Initial Java Card DL proof goal.

$\{\mathtt{i} := 2 \,||\, \mathtt{j} := 2\}\langle\rangle \mathtt{i} \doteq \mathtt{j}$, which would simplify to $\{\mathtt{i} := 2 \,||\, \mathtt{j} := 2\}\mathtt{i} \doteq \mathtt{j}$, and evaluate to *true*. This can be seen as symbolic execution, with the updates tracking all state changes. The state changes are kept within the curly brackets as a set of pairs of program locations and side-effect-free expressions, called *updates*. The updates can be seen as delayed substitutions (but there is much more to updates [25]). The calculus takes care of complications such as aliasing, exceptions, side-effects in branching conditions etc; and transfers those to side-effect free expressions in the updates, all without destroying the syntactic structure of the postcondition. This is a difference to other systems and something that we take advantage of in our analysis.

The system that we are using is called KeY [3] and the logic Java Card Dynamic Logic [4, 6]. KeY is a system for integrated design, development and formal verification of Java Card programs. In particular it has an interactive theorem prover for Java Card DL with the above properties.

3 Example

In this section we illustrate some of the problems encountered when using induction for proving the total correctness of a loop and at the same time we show our method at work. We wish to prove that the program in Figure 1 terminates in a state where `s` contains the product of the values that `i` and `j` had before the loop. That of course can only be proven under some assumptions about the initial values of `i`, `j` and `s`. We specify these requirements using a precondition and a postcondition, see Figure 2. In the precondition we express that the value of `i` in the starting state should be positive. The postcondition says that the value of `s` is the product of the starting values of `i` and `j`. In order to initialise `s` with $\mathtt{s} := 0$, we can either modify the program or we can modify the description of the starting state. Note that the *logical variables* *il* and *jl* are introduced since we cannot quantify over *program variables* (`i` and `j`).

In the general inductive proving process [27] there are at least three crucial points of interaction: 1) supplying the induction variable, 2) the choice of induction rule, and 3) supplying the induction formula. The last point is the focus of this paper, but we give a brief account of the entire process. First, to simplify the initial proof obligation (Figure 2) we use the rule allRight (see Appendix) to replace the universally quantified variables with Skolem constants il_c and jl_c and the rule impRight to decompose the implication. Applications of these two rules, and many others, such as commutativity and associativity, are performed automatically by the KeY prover. We will make them tacitly from now on.

The terminating condition for the loop tells us that the only suitable option for the induction variable is `i`. We expect the loop to terminate in the induction base case, that is, when the induction variable is zero. Again, we need a logical variable to quantify over, so we introduce

$\phi(il) \leftrightarrow$
$\forall sl.(0 \leq il \rightarrow$
$\{\texttt{i} := il \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := sl\}$
$\langle$ `while (0 < i) {`
`  i--;`
`  s = s + j;`
`}` $\rangle\ POST(\texttt{s}, il, jl_c, sl))$

Figure 3: Induction formula guess. Note that we have added a *meta-variable POST* to denote the unknown postcondition formula. We have also added a universally quantified variable sl so that we can later make the state descriptions syntactically equal.

$\forall il.\forall sl.(0 \leq il \rightarrow$
$\{\texttt{i} := il \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := sl\}$
$\langle$ `while (0 < i) {`
`  i--;`
`  s = s + j;`
`}` $\rangle\ POST(\texttt{s}, il, jl_c, sl))$,
$0 \leq il_c$
$\Longrightarrow$
$\{\texttt{i} := il_c \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := 0\}$
$\langle$ `while (0 < i) {`
`  i--;`
`  s = s + j;`
`}` $\rangle\ \texttt{s} \doteq il_c * jl_c$

Figure 4: Use case proof goal.

il to be the induction variable. We make sure that `i` and il are connected in the initial state description of the induction formula $\phi(il)$. The update to the induction variable is given by `i--;` so the standard natInduct rule is suitable. The increment of the induction variable in the step case of natInduct $\forall n.0 \leq n \wedge \phi(n) \rightarrow \phi(n+1)$ will then be cancelled by the decrement of the induction variable when symbolically executing the loop, thus removing one obstacle to closing the proof.

Trivial Induction Formula—Failed Proof Attempt After having chosen the induction variable il and natInduct as the induction rule it remains to supply the induction formula $\phi(il)$. The simplest possible approach would be to supply the simplified initial proof obligation modified only by replacing il_c with the induction variable il, the same formula as in Figure 2. The step case of a proof attempt using natInduct would (after skolemisation) be $\phi(il_c) \rightarrow \phi(il_c+1)$. After having unrolled one iteration of the loop body in the succedent, the side effects would be collected in the state description. Since we cannot make the states in the antecedent $\{\texttt{i} := il_c \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := 0\}$ and in the succedent $\{\texttt{i} := il_c \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := jl_c\}$ syntactically equal, the proof attempt is stuck. Furthermore, the postcondition in the antecedent $\texttt{s} \doteq il_c * jl_c$ and in the succedent $\texttt{s} \doteq (il_c + 1) * jl_c$ cannot be made syntactically equal. At this point we are forced to start the induction proving process over with a better induction formula.

Generalised Induction Formula Guess Our idea is to construct a suitable induction formula without a number of tedious proof attempts but instead making use of the structure of the proof method. We first construct a *guess* for the induction formula $\phi(il)$, see Figure 3. The idea now is to perform the induction process as usual and learn the constraints on the correct postcondition. We first generate constraints for the use case, since the original proof obligation is a hard constraint.

Constraints from the Use Case After instantiating the use case with our induction formula guess we have the formula in Figure 4. To make the updates of the antecedent and succedent syntactically equivalent, we supply the substitution $[il/il_c, sl/0]$. To close this proof branch it only remains to show that the postconditions are syntactically equivalent, or that the guessed postcondition implies the postcondition of the starting proof obligation (under the assumption

$\forall sl.(0 \leq il_c \rightarrow$
$\{\texttt{i} := il_c \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := sl\}$
$\langle$ `while (0 < i) {`
`i--;`
`s = s + j;`
`}` $\rangle\, POST(\texttt{s}, il_c, jl_c, sl))$,
$0 \leq il_c + 1$
$\Longrightarrow$
$\{\texttt{i} := il_c + 1 \,||\, \texttt{j} := jl_c \,||$
$\texttt{s} := sl_c\}$
$\langle$ `while (0 < i) {`
`i--;`
`s = s + j;`
`}` $\rangle\, POST(\texttt{s}, il_c + 1, jl_c, sl_c)$

Figure 5: Step case proof goal.

$\forall sl.(0 \leq il_c \rightarrow$
$\{\texttt{i} := il_c \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := sl\}$
$\langle$ `while (0 < i) {`
`i--;`
`s = s + j;`
`}` $\rangle\, POST(\texttt{s}, il_c, jl_c, sl))$,
$0 < il_c + 1$
$\Longrightarrow$
$\{\texttt{i} := il_c + 1 - 1 \,||\, \texttt{j} := jl_c \,||$
$\texttt{s} := sl_c + jl_c\}$
$\langle$ `while (0 < i) {`
`i--;`
`s = s + j;`
`}` $\rangle\, POST(\texttt{s}, il_c + 1, jl_c, sl_c)$

Figure 6: Step case, loop entered.

$0 \leq 0$
$\Longrightarrow$
$\{\texttt{i} := 0 \,||\, \texttt{j} := jl_c \,||\, \texttt{s} := sl_c\}$
$\langle$ `while (0 < i) {`
`i--;`
`s = s + j;`
`}` $\rangle\, POST(\texttt{s}, 0, jl_c, sl_c)$

Figure 7: The base case proof obligation corresponds to termination of the loop.

of the starting precondition). Since the postcondition is an unknown we generate the first constraint:

$$POST(\texttt{s}, il_c, jl_c, 0) \land 0 \leq il_c \rightarrow \texttt{s} \doteq il_c * jl_c \tag{E1}$$

Constraints from the Step Case We instantiate the step case with our induction formula guess and get the formula in Figure 5. After symbolic execution of the succedent, the proof branches at the while condition. In one branch the loop terminates, the program is of no more help, $il_c + 1$ must be zero and we get the following constraint:

$$POST(sl_c, 0, jl_c, sl_c) \tag{E2}$$

In the other branch (Figure 6), the loop is entered and the side effect of the loop body is visible in the update. Since an appropriate combination of induction rule and induction variable has been chosen, the effect on the induction variable is cancelled by the induction step. With the substitution $[sl/sl_c + jl_c]$ we achieve syntactic equivalence in the updates and we get rid of the quantifier. After impLeft and simplification we can prove the precondition branch $0 < il_c + 1 \rightarrow$

$0 \leq il_c$ and the remaining constraint for the postcondition is

$$0 \leq il_c \wedge POST(\mathtt{s}, il_c, jl_c, sl_c + jl_c) \rightarrow POST(\mathtt{s}, il_c + 1, jl_c, sl_c) \tag{E3}$$

Constraints from the Base Case Finally, let us consider the base case branch, see Figure 7. Since the loop condition is false when the induction variable is 0, we get the following constraint:

$$POST(sl_c, 0, jl_c, sl_c) \tag{E4}$$

3.1 Solution to the Constraints

We have generated one constraint from the use case (E1), one constraint from the base case (E4) (which is the same as the constraint from the terminating branch of the step case (E2)) and one constraint from the loop-entering branch of the step case (E3). We can solve these constraints (by hand) to the following which will yield a successful proof:

$$POST(\mathtt{s}, il, jl, sl) \leftrightarrow \mathtt{s} \doteq il * jl + sl$$

4 Deriving Constraints from a Symbolic Proof Attempt

We have shown by example how to construct an induction formula guess and how to use a structured induction proving process to generate the constraints that must be satisfied for a successful proof. We will now describe the method in more general terms, including a symbolic proof attempt.

1. The method has the original proof obligation as the starting point. First, basic simplification is performed. Universally quantified variables are replaced by arbitrary constants, so called Skolem constants, using the rule allRight. Implications are simplified using the rule impRight.

2. The next step is to supply the induction variable and the induction rule. The induction variable is found by observing the terminating condition of the loop. The induction rule is constructed to suit the particular program at hand by analysing the state change to the induction variable after symbolic execution of one loop iteration. For details on customised induction rules, see [13, 22]. In this paper we assume that the induction variable and a suitable induction rule are given, and instead focus on the remaining point: generalisation of the induction formula.

3. As a first step in generalising the induction formula, we make a naive proof attempt where we supply the original proof obligation as the induction formula. As prescribed by the induction proving process in [27], in the step case of the inductive proof attempt we use the rule loopUnwind to symbolically execute one loop iteration. Then we use allLeft to instantiate any universally quantified variables in such a way that the states of the antecedent and succedent become syntactically equal for as many program variables as possible. From the remaining proof goal we learn how to construct a better induction formula.

4. Now we create a guess for the induction formula based on the information from the failed naive proof attempt. For every program variable that has a syntactic mismatch between

$$
\begin{array}{l}
\Longrightarrow \\
\forall \overrightarrow{lv}.(pre_{uc}(\overrightarrow{lv}) \rightarrow \\
\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_{uc}}(\overrightarrow{lv})\} \\
\langle \texttt{ while } (c_{while}) \texttt{ \{} \\
\quad \texttt{loop body;} \\
\texttt{ \}} \rangle \, post_{uc}(\overrightarrow{\texttt{pv}}, \overrightarrow{lv}))
\end{array}
$$

Figure 8: General proof goal.

$$
\begin{array}{l}
\forall \overrightarrow{gv}.(PRE_{\phi}(il, \overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_{\phi}}(il, \overrightarrow{gv})\} \\
\langle \texttt{ while } (c_{while}) \texttt{ \{} \\
\quad \texttt{loop body;} \\
\texttt{ \}} \rangle \, POST_{\phi}(\overrightarrow{\texttt{pv}}, il, \overrightarrow{gv}))
\end{array}
$$

Figure 9: General induction formula guess.

the antecedent and succedent, we introduce a universally quantified variable. Furthermore, we introduce a meta-variable that is a placeholder for the correct but unknown postcondition formula. This depends on the previously used variables and the new universally quantified variables.

5. Then we start a new proof attempt, this time with the induction formula guess. We proceed with our structured induction proving process. In every open proof branch, there is a simplified proof goal that cannot be closed unless the meta-variables are instantiated. Thus every branch contributes one constraint on the meta-variables. If the constraints can be solved, we have found a correct generalisation of the induction formula and the proof is done.

4.1 Initial Proof Obligation

The original proof obligation ϕ_{pog} is of a general form that can be seen in Figure 8, where $\overrightarrow{lv}$ is a set of logical variables and $\overrightarrow{\texttt{pv}}$ a set of program variables. There is a precondition pre_{uc}, a postcondition $post_{uc}$, and updates $\overrightarrow{\mu_{uc}}(\overrightarrow{lv})$ that describe the state of the program variables in the use case, in terms of the values of the logical variables.

After running basic simplifying heuristics and applying allRight, the logical variables in the succedent are replaced with skolem constants $\overrightarrow{lv_c}$. At this point a non-trivial proof attempt will be stuck and require the application of induction. Provided that we have found the right induction rule and induction variable, the most challenging task is to find the right induction formula.

4.2 Induction Formula Guess

Assume that—by an initial analysis of a trivial proof attempt—we know which variables that need to be generalised, we know the correct induction variable, how the induction variable is related to the program variables by a suitable update, and which induction rule to use. This part of our guess is given by [27, 22]. Then we make a general guess for the induction formula, $\phi(il)$, which is stated in Figure 9. The guess follows the same pattern as the proof obligation itself (Figure 8). The modifications consist of the addition of an induction variable, possibly additional generalised logical variables $\overrightarrow{gv}$, and updates $\overrightarrow{\mu_{\phi}}$ to relate these new logical variables to the program variables. Note that if we introduce additional universally quantified variables, then the generalised variable set $\overrightarrow{gv}$ is different from the logical variables $\overrightarrow{lv}$ in the original proof obligation (Figure 8). For example, in the product example in Section 3 the starting set of variables $\overrightarrow{lv}$ was il, jl and the generalised set of variables $\overrightarrow{gv}$ was il, jl, sl. The most important difference between the original proof obligation (Figure 8) and the guess (Figure 9)

$\forall il.\forall \overrightarrow{gv}.(PRE_\phi(il, \overrightarrow{gv}) \rightarrow$
$\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}(il, \overrightarrow{gv})\}$
$\langle$ `while (`c_{while}`) {`
`loop body;`
`}` $\rangle$ $POST_\phi(\overrightarrow{\texttt{pv}}, il, \overrightarrow{gv}))$,
$pre_{uc}(\overrightarrow{lv_c})$
$\Longrightarrow$
$\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_{uc}}(\overrightarrow{lv_c})\}$
$\langle$ `while (`c_{while}`) {`
`loop body;`
`}` $\rangle$ $post_{uc}(\overrightarrow{\texttt{pv}}, \overrightarrow{lv_c})$

Figure 10: Use case proof goal.

$\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}([\sigma_{uc}]il, [\sigma_{uc}]\overrightarrow{gv})\}$
$\langle$ `while (`c_{while}`) {`
`loop body;`
`}` $\rangle$ $POST_\phi(\overrightarrow{\texttt{pv}}, [\sigma_{uc}]il, [\sigma_{uc}]\overrightarrow{gv})$,
$pre_{uc}(\overrightarrow{lv_c})$
$\Longrightarrow$
$\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_{uc}}(\overrightarrow{lv_c})\}$
$\langle$ `while (`c_{while}`) {`
`loop body;`
`}` $\rangle$ $post_{uc}(\overrightarrow{\texttt{pv}}, \overrightarrow{lv_c})$

Figure 11: Instantiated and simplified goal, postcondition branch.

is the presence of $POST_\phi$, the yet unknown postcondition. We have introduced/selected $\overrightarrow{gv}$, il etc., in such a way that we know that the proof can be closed later, provided that we find a suitable $POST_\phi$. To turn the entire postcondition into an unknown is often too large a guess to be solvable. In the examples in the next section we will show some ways to make the guess more local. Since a different precondition is often necessary, we introduce a meta-variable PRE_ϕ for it.

After making this guess for the induction formula we will proceed with the induction proving process. If we can prove that the generalised induction formula is valid, and that this formula is enough to prove the original proof obligation, then we can close the proof. Let us follow the induction proving process and generate the constraints for $POST_\phi$ and PRE_ϕ.

4.3 Deriving Constraints from the Use Case

In this branch the task is always to prove that the induction formula (Figure 9) implies the original proof obligation (Figure 8). It is here that we apply the generalised induction formula which is still unknown and to be proved in the other branches of the induction proof. The use case branch (Figure 10) is the most constraining branch since it contains the problem as supplied by the user.

To generate the use case constraints, we first need to find a substitution σ_{uc} such that each update in $\overrightarrow{\mu_\phi}([\sigma_{uc}]il, [\sigma_{uc}]\overrightarrow{gv})$ is syntactically equal to the corresponding update in $\overrightarrow{\mu_{uc}}(\overrightarrow{lv_c})$. Usually it is easy to find such a σ_{uc}. Then we use this substitution to instantiate il and $\overrightarrow{gv}$. After the instantiation and some simplification, the process has given one branch for the precondition, and one for the postcondition (Figure 11). From the precondition branch, we can generate a constraint (C1). If a solution for the constraint cannot be found, the constraint can be used to inform the user that the precondition that he has supplied is too weak, which is a common error.

$$pre_{uc}(\overrightarrow{lv_c}) \rightarrow PRE_\phi([\sigma_{uc}]il, [\sigma_{uc}]\overrightarrow{gv}) \tag{C1}$$

Next we consider the postcondition branch of the use case (Figure 11). Since σ_{uc} has been chosen to make the updates syntactically equal and the programs are the same, only the postcondition remains to be shown. Thus we generate a constraint on the postcondition (C2).

$$POST_\phi(\overrightarrow{\texttt{pv}}, [\sigma_{uc}]il, [\sigma_{uc}]\overrightarrow{gv}) \wedge pre_{uc}(\overrightarrow{lv_c}) \rightarrow post_{uc}(\overrightarrow{\texttt{pv}}, \overrightarrow{lv_c}) \tag{C2}$$

$$
\begin{array}{l}
\forall \overrightarrow{gv}.(PRE_\phi(il_c, \overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c, \overrightarrow{gv})\} \\
\langle\ \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\texttt{\} } \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}}, il_c, \overrightarrow{gv}))\ , \\
PRE_\phi(il_c + 1, \overrightarrow{gv_c}) \\
\Longrightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c + 1, \overrightarrow{gv_c})\} \\
\langle\ \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\texttt{\} } \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}}, il_c + 1, \overrightarrow{gv_c})
\end{array}
$$

Figure 12: Step case goal.

$$
\begin{array}{l}
\forall \overrightarrow{gv}.(PRE_\phi(il_c, \overrightarrow{gv}) \rightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c, \overrightarrow{gv})\} \\
\langle\ \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\texttt{\} } \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}}, il_c, \overrightarrow{gv})), \\
PRE_\phi(il_c + 1, \overrightarrow{gv_c}), \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c + 1, \overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \\
\Longrightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_{sym}}(il_c + 1, \overrightarrow{gv_c})\} \\
\langle\ \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\texttt{\} } \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}}, il_c + 1, \overrightarrow{gv_c})
\end{array}
$$

Figure 13: Step case, loop entered.

$$
\begin{array}{l}
PRE_\phi(0, \overrightarrow{gv_c}) \\
\Longrightarrow \\
\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(0, \overrightarrow{gv_c})\} \\
\langle\ \texttt{while (}c_{while}\texttt{) \{} \\
\quad \texttt{loop body;} \\
\texttt{\} } \rangle\ POST_\phi(\overrightarrow{\mathtt{pv}}, 0, \overrightarrow{gv_c})
\end{array}
$$

Figure 14: Base case proof obligation.

4.4 Deriving Constraints from the Step Case

The step case (Figure 12) holds two instances of the induction formula. First we symbolically execute the succedent of the proof obligation, which makes the proof branch at the while condition c_{while}. Recall that a negative while condition corresponds to termination of the loop so after symbolic execution the program has disappeared and the remaining constraint for this branch is (C3).

$$
\begin{array}{l}
PRE_\phi(il_c + 1, \overrightarrow{gv_c})\ \wedge \neg\{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c + 1, \overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \\
\rightarrow \{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c + 1, \overrightarrow{gv_c})\}\ POST_\phi(\overrightarrow{\mathtt{pv}}, il_c + 1, \overrightarrow{gv_c})
\end{array}
\tag{C3}
$$

The case where the while condition is positive, and in the succedent the loop body has been executed once with the help of loopUnwind (Figure 13), gives rise to the new update $\overrightarrow{\mu_{sym}}$. The induction variable and the induction rule have already been chosen so that for the program variable that depends on the induction variable il, $\overrightarrow{\mu_{sym}}(il_c + 1)$ is syntactically equivalent to $\overrightarrow{\mu_\phi}(il_c)$. Also here in the step case we must find a substitution σ_{sc} so that $\overrightarrow{\mu_{sym}}(il_c + 1, \overrightarrow{gv_c})$ is syntactically equivalent to $\overrightarrow{\mu_\phi}(il_c, [\sigma_{sc}]\overrightarrow{gv})$ for the rest of the program variables. Then we use this substitution to instantiate $\overrightarrow{gv}$. After applying implLeft we get another branching, with one goal for the precondition and one goal for the postcondition. The first one gives another constraint on the precondition:

$$
PRE_\phi(il_c + 1, \overrightarrow{gv_c})\ \wedge \{\overrightarrow{\mathtt{pv}} := \overrightarrow{\mu_\phi}(il_c + 1, \overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\mathtt{pv}}) \rightarrow PRE_\phi(il_c, [\sigma_{sc}]\overrightarrow{gv}) \tag{C4}
$$

$\forall nl.(0 \leq nl \rightarrow$
$\{\texttt{n} := nl\}$
$\langle$ `i=0;`
`r=0;`
`while (i < n) {`
`i++;`
`r = r+(i*i*i);`
`}` $\rangle\ 4\texttt{r} \doteq nl^2(nl+1)^2)$

Figure 15: Cubic sum: Original proof obligation.

$\forall rl.(0 \leq nl_c \rightarrow$
$\{\texttt{i} := nl_c - kl \,||\, \texttt{n} := nl_c \,||\, \texttt{r} := rl\}$
$\langle$ `while (i < n) {`
`i++;`
`r = r+(i*i*i);`
`}` $\rangle\ POST(\texttt{r}, kl, rl, nl_c))$

Figure 16: Cubic sum: Induction formula guess.

From the other branch we can generate a constraint on the postcondition:

$$PRE_\phi(il_c+1, \overrightarrow{gv_c}) \wedge \{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}(il_c+1, \overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\texttt{pv}}) \wedge POST_\phi(\overrightarrow{\texttt{pv}}, il_c, [\sigma_{sc}]\overrightarrow{gv}) \rightarrow POST_\phi(\overrightarrow{\texttt{pv}}, il_c+1, \overrightarrow{gv_c}) \quad \text{(C5)}$$

4.5 Deriving Constraints from the Base Case

At this point, we prepare for one more constraint that corresponds to termination of the loop, see Figure 14. We have already decided upon an induction variable and updates such that the while condition c_{while} is false and the loop terminates when $il = 0$. So symbolic execution of the program in Figure 14 will generate an empty program and this gives us the last constraint:

$$PRE_\phi(0, \overrightarrow{gv_c}) \wedge \neg\{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}(0, \overrightarrow{gv_c})\}\ c_{while}(\overrightarrow{\texttt{pv}}) \rightarrow \{\overrightarrow{\texttt{pv}} := \overrightarrow{\mu_\phi}(0, \overrightarrow{gv_c})\}\ POST_\phi(\overrightarrow{\texttt{pv}}, 0, \overrightarrow{gv_c}) \quad \text{(C6)}$$

Now it only remains to solve the constraints. In this paper we do not present a method for solving the constraints but a discussion of ideas for constraint solving is included in Section 7.

5 Example: Cubic Sum

Let us try our method at a program that computes the so called cubic sum.

$$\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4\ .$$

The problem would be specified in JAVA CARD DL as in Figure 15. We introduce an induction variable kl, where $kl = \texttt{n} - \texttt{i}$ (see the terminating condition of the loop), and we choose to apply the normal natInduct rule (see the update to the induction variable inside the loop). Then we construct the induction formula guess in Figure 16.

We set the guess for the precondition PRE to *true*. We then generate the following constraints, from the use case, the step case and the base case respectively:

$$POST(\texttt{r}, nl_c, 0, nl_c) \ \rightarrow\ 4\texttt{r} \doteq nl_c^2(nl_c+1)^2 \quad \text{(CS1)}$$

$$POST(\texttt{r}, kl_c, rl_c + (nl_c - kl_c)^3, nl_c) \ \rightarrow\ POST(\texttt{r}, kl_c+1, rl_c, nl_c) \quad \text{(CS2)}$$

$$POST(rl_c, 0, rl_c, nl_c) \quad \text{(CS3)}$$

$\Longrightarrow$
$\langle$ `m = 0;`
`x = 0;`
`while (x < a.length) {`
`if (a[x] < a[m]) {`
`m = x;`
`}`
`x++;`
`}` $\rangle$ $\forall kl.0 \leq kl <$ `a.length`
$\rightarrow$ `a[m]` $\leq$ `a[`kl`]`)

Figure 17: Findmin: Original proof obligation.

$\forall ml.(PRE(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}) \rightarrow$
$\{\texttt{x} := \texttt{a.length} - il \,||\, \texttt{m} := ml\}$
$\langle$ `while (x < a.length) {`
`if (a[x] < a[m]) {`
`m = x;`
`}`
`x++;`
`}` $\rangle$ $\forall kl.(F_1(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}) \leq kl$
$< F_2(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length})$
$\rightarrow P(\texttt{x}, \texttt{m}, kl, il, ml, \texttt{a.length}))$
$\wedge Q(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}))$

Figure 18: Findmin: Induction formula guess.

Now we can get hold of the desired postcondition by solving the constraints. The constraints are satisfied by the following $POST(\texttt{r}, kl, rl, nl_c)$, which makes the induction formula guess (Figure 16) provable:

$$POST(\texttt{r}, kl, rl, nl_c) \leftrightarrow 4(\texttt{r} - rl) \doteq nl_c^2(nl_c + 1)^2 - (nl_c - kl)^2(nl_c - kl + 1)^2 \ .$$

6 Example: Find the Minimal Element in an Array

Now let us consider a more advanced example, finding the index of the smallest element in an array, see Figure 17. The postcondition of this loop is more complicated than in the previous examples, as it involves quantification over an array interval. In the following constraint generation, we will make the "guess" more local than before by introducing several meta-variables in the postcondition. This approach can be seen as an attempt at classifying the postcondition based on its structure. In this case the postcondition is that a property holds for an interval.

Induction Formula Guess We start by choosing an appropriate induction variable. The `while` condition suggests that the loop terminates when `a.length` $-$ `x` is zero, so we introduce the induction variable il = `a.length` $-$ `x` and specify the relation between il and `x` in the update of `x`. The induction variable will be decreased by one in every loop iteration (via `x++`) so natInduct will be a suitable induction rule.

In the induction formula guess (Figure 18), we could put one meta-variable $POST$ as a guess for the entire postcondition as described in Section 4. However, in our quest to generate constraints that are easier to solve, and to exploit the input from the user as much as we can, we make a slightly different guess. The idea is to use the structure of the postcondition that comes from the proof obligation given by the user. We keep the general shape, in this case quantification over a range, and modify it by replacing the "leaves" with meta-variables. In addition to the part that comes from the original proof obligation, we add another unknown (here Q), to allow further strengthening of the postcondition. With a guess like this we can treat a whole class of postconditions where the user wants a certain property to hold for all elements of an array. We also introduce a meta-variable PRE for the precondition. We then perform a proof attempt in order to generate the constraints. The proof attempt and the generated constraints have been omitted for brevity, but they can be found in [28].

Solution to the Constraints The below solution to the constraints was derived by hand. See Section 7 for a discussion on how this solution could possibly be found automatically.

$$F_1(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}) = \texttt{a.length} - il \quad \text{(FM1)}$$
$$F_2(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}) = \texttt{a.length} \quad \text{(FM2)}$$
$$P(\texttt{x}, \texttt{m}, kl, il, ml, \texttt{a.length}) \leftrightarrow \texttt{a[m]} \leq \texttt{a}[kl] \quad \text{(FM3)}$$
$$Q(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}) \leftrightarrow \texttt{a[m]} \leq \texttt{a}[ml] \quad \text{(FM4)}$$
$$PRE(\texttt{x}, \texttt{m}, il, ml, \texttt{a.length}) \leftrightarrow true \quad \text{(FM5)}$$

7 Constraint Solving

The method described in this article generates an induction formula guess with unknown parts, and constraints on those parts. If a solution for the constraints can be found, the resulting formula can be automatically proven by induction. The author solves by hand constraints like the ones in the examples in Sections 5 and 6 roughly according to this iterative recipe:

1. Pick values for all of the meta-variables to satisfy the constraints from the use case. The use case is always considered first because it contains a lot of concrete information about the problem as given by the user. It also only has one instance of the induction formula, making it easier to solve by purely syntactic methods. We try to include the induction variable so that, intuitively, the postcondition holds from the current value of the induction variable until the loop terminates.

2. Next we consider the step case postcondition constraint(s), because the step case is the heart of an induction proof. If the antecedent does not completely imply the succedent with the current meta-variable values, the values are amended. For example, in Section 6, the range over which the succedent quantifies includes one element more than the range in the antecedent. Thus we may need to amend the solution to account for that element. If so, we have changed the postcondition and we start over, with the new meta-variable instantiation. If there is basic type information missing, try to add it to the precondition.

3. Consider the constraint from the base case. In the process, the base case is mostly a sanity check. If the solution under consideration does not satisfy this constraint, we often need to rethink how the induction variable is used and go back to the use case.

4. If the precondition instantiation is not *true*, check all the constraints on the precondition.

This process is largely guided by trying to achieve syntactic equivalence wherever possible. For example, we can first try to solve an implication by looking for syntactic equality. Other tricks include splitting ranges, and weakening by removing conjuncts and shrinking ranges. This constraint solving method is guided by experience and intuition, and does not lend itself to direct implementation. However, it may be used as a starting point in optimising the search order in a mechanical solver.

An automatic method for solving the generated constraints is beyond the scope of this article. Indeed, since the constraints involve second-order variables, solving them is undecidable in the general case. However, as with many undecidable problems, there are partial algorithms for solving second-order constraints. If a given set of constraints has a solution where all meta-variables are instantiated with values in some known limited class, we can often solve them automatically. One example of such a class is polynomials (for integer-valued meta-variables)

and polynomial equations (for truth-valued meta-variables), of some bounded degree. Such guesses have been used to solve constraints in loop invariant generation [26, 17]. Another interesting class is linear terms and linear inequalities. Gulwani et al. [12] convert second-order constraints for program analysis to SAT problems, replacing second-order meta-variables with linear inequalities.

The choice of the solution class to try could be guided by the original proof obligation as given by the user. For example, in the product example in Section 3, the original postcondition was a polynomial equation of degree 2 and the constraints could be solved with a polynomial equation of the same degree. In the cubic sum example in Section 5, both the user-supplied postcondition and the generated solution was a polynomial equation of degree 4. Likewise, in the findmin example in Section 6, the original proof obligation contained a linear inequality, as did the solution. Of course, if no solution is found in one class, other classes could be tried. If no solutions can be found at all, we could go back and try to refine the induction formula guess. In general, we can treat the constraint solving as a search problem, where we make progressively more refined guesses when simpler ones fail.

8 Limitations and Future Work

This article outlines an idea for a constraint-based approach to induction formula generalisation. There is plenty more to do before this is a practical and automatic method. Firstly, there is no implementation of the constraint generation method, though it should be rather straightforward to build a proof-of-concept implementation within the KeY system. Secondly, we have not tried to solve the generated constraints automatically, although the previous section presents some ideas for how that could be done.

The examples that we have shown outline classes of problems that can be handled by our method. It would be interesting to try to describe these classes formally, and to identify additional classes of problems. This would naturally also include trying a large number of additional examples.

The calculus that we have used, Java Card DL, can handle more complex language features than we have used in the examples here. It would be interesting to try our method on examples that include array modification and exceptions. In the case of array modification, quantified updates [25] could probably be used to achieve syntactic equivalence in the updates of the array elements.

So far, we have only considered single (unnested) loops. Perhaps the method could be generalised to handle nested loops by generating (and solving) constraints for all the loops simultaneously. Furthermore, we have only dealt with induction over natural numbers, though it should be possible to extend the method to structural induction.

9 Related Work

The automation of inductive proofs has been a research topic for more than 30 years. Several systems for reasoning about programs in functional programming languages have induction heuristics, most notably ACL2 [8, 7], Verifun [29, 1], RRL [18], and Rippling [9].

ACL2 has a powerful mechanism for generating induction schemes from recursive definitions. However, the mechanism for induction formula generalisation is simpler. The ACL2 generalisation heuristic detects common subterms in the hypothesis and conclusion of the induction step,

after unrolling the function in the conclusion and simplifying it, and then constructs a generalised induction formula by replacing the subterms with new universally quantified variables. For example, ACL2 does not automatically prove the correctness of the cubic sum example in Section 5 (translated to LISP) as it is originally stated. Thanks to the mature and widely available implementation of ACL2 it was easy to translate the cubic sum example to ACL2 and attempt to prove it correct. We translate the loop to a tail recursive function with an accumulator, implement the closed-form solution as another function, and try to prove that the two give the same result for any natural number:

```
(defun csum1 (i s) (if (not (zp i)) (csum1 (- i 1) (+ s (* i i i))) s))
(defun csum2 (n) (* n n (+ n 1) (+ n 1) 1/4))
(defthm csum-correct (implies (natp n) (equal (csum1 n 0) (csum2 n))))
```

However, ACL2 is not able to prove this. But if we instead use the generalized formula produced by our method, the proof succeeds (see [16] for related work on generalisation with accumulators):

```
(defthm csum-correct-gen
  (implies (and (natp n) (natp s)) (equal (csum1 n s) (+ (csum2 n) s))))
```

Rippling is an induction heuristic for guiding rewriting (proofs) by annotating both the formulae (proof goals) and the rewrite rules so that only those rewrites are allowed that actually make the goal syntactically closer to the assumption and in the end hopefully closable. The annotations can be used to derive suitable induction schemes, to create intermediate lemmas and to create induction formula generalisations. Our approach has several similarities to the rippling approach, most importantly that it is syntactically driven and that it analyses failed proof attempts. We were inspired by their use of meta-variables in the generalisation process.

A heuristic for lemma discovery in the rewrite-based induction theorem prover RRL [18], also analyses failed proof attempts. Their introduction of "non-induction variables" seems to correspond to our introduction of new universally quantified variables. A difference is that they need to search for the right instantiations for those variables, whereas that part is rather straightforward and syntactically driven in our approach, thanks to the update mechanism of JAVA CARD DL.

`VeriFun` [31] is a semi-automatic verification tool for functional programs written in the $\mathcal{L}$ [30] programming language. Its induction formula generalisation heuristics have been developed in a long line of research [29, 15, 1]. The heuristics are inverse applications of sound inferences. Like in ACL2, repeated subterms are replaced by new variables (*inverse substitution*). Other heuristics include *inverse substitution* to generalise apart multiple occurrences of a variable, *inverse weakening* to remove conditions, and *inverse functionality* to remove function applications. Aderhold [1] recently extended `VeriFun` to use a fast disprover [2] for detecting the over-generalisations that each of the heuristics may produce.

Several techniques for the automatic generation of loop invariants have been developed using abstract interpretation. In [20] a widening operator is built into an SMT solver, so that the widening operator can be applied by the solver whenever it needs to approximate loop invariants. In their previous approach [21], a separate static analyser and a theorem prover were used together so that loop invariants could be refined using counterexamples from the theorem prover. In [23] program statements are translated into transformations on polynomial ideals and a sound and complete method for automatically finding polynomial loop invariants by computing Gröbner bases is then given. The approach has the restriction that it must be possible to abstract conditionals in the program to polynomial equalities and disequalities, but

it can solve many non-trivial examples. The algebraic foundations of inferring conjunctions of polynomial equations as invariants by computing Gröbner bases is further described in [24]. [19] also presents an algebraic method for finding polynomial equations as loop invariants. Another approach that generates constraints based on guessing that the invariant is a polynomial with real coefficients is presented in [26]. It also uses Gröbner bases to solve the constraints.

A very recent approach [12] shows a constraint-based way to perform program analysis. From the control-flow graph, constraints on loop invariants are generated, where the loop invariant itself is a second-order meta-variable. The constraints are transformed to a form that can be handled by an off-the-shelf SAT solver, by replacing the second-order variables with boolean combinations of linear inequalities over program variables. The approach in [17] also generates constraints where the loop invariant is a second-order meta-variable. The loop invariant is then hypothesised to be a polynomial equation and the constraints are solved by algebraic techniques. It would be interesting to use the constraint solving techniques from both of these approaches to solve the constraints generated in our approach.

10 Summary and Conclusions

We have presented an idea for mechanical generalisation of induction formulae. It introduces meta-variables in an existing proof obligation that has been provided by the user. It then uses a well-structured and syntactically driven process, that could be called a symbolic proof attempt, to generate constraints on the meta-variables. The method is based on a weakest-precondition calculus with forward symbolic execution that does not destroy the syntactic structure of the postcondition. Since the calculus that we use can transform programming language features with side-effects to side-effect-free formulae, we take advantage of this to handle a complex imperative language, JAVA CARD.

We have not presented a method for solving the constraints that our method generates. However, we have discussed existing work on solving classes of constraints similar to those that we generate. Furthermore, when solving our constraints by hand, we have made some observations that could be useful for an automatic constraint solver.

Most of the existing work on automatic verification of data correctness for loops is done in the context of a small (functional or imperative) language. In contrast, our approach is based on a logic which can directly handle a large imperative and object-oriented language. While it is true that any program in a large programming language can be translated to a program in a small language, this can make it much harder to give sensible feedback to the verification tool user. An important piece of future work would be to implement the method and evaluate it on a number of non-trivial examples. Hopefully, we can make use of the mechanism in KeY to replay proofs (attempts) [5] in the implementation.

Acknowledgements The author would like to thank Björn Bringert, Koen Claessen, John Hughes, Reiner Hähnle, Deepak Kapur, Philipp Rümmer, Mary Sheeran, and the anonymous reviewers, for their help and comments on earlier drafts of this paper.

References

[1] Markus Aderhold. Improvements in formula generalization. In Frank Pfenning, editor, *Proceedings of the 21st Conference on Automated Deduction (CADE-21)*, volume 4603 of *Lecture Notes in Artificial Intelligence*, pages 231–246, Bremen, Germany, 2007. Springer-Verlag.

[2] Markus Aderhold, Christoph Walther, Daniel Szallies, and Andreas Schlosser. A fast disprover for verifun. In Wolfgang Ahrendt, Peter Baumgartner, and Hans de Nivelle, editors, *Proceedings of the 3rd Workshop on Disproving, IJCAR 2006*, pages 59–69, Seattle (WA), USA, 2006.

[3] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4(1):32–54, February 2005.

[4] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[5] Bernhard Beckert and Vladimir Klebanov. Proof reuse for deductive program verification. In J. Cuellar and Z. Liu, editors, *Software Engineering and Formal Methods (SEFM), Beijing, China*. IEEE Press, 2004.

[6] Bernhard Beckert and Bettina Sasse. Handling JAVA's abrupt termination in a sequent calculus for Dynamic Logic. In *IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14, 2001.

[7] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.

[8] Robert Boyer and J. Strother Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.

[9] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.

[10] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Co., Amsterdam, 1969. Edited by M. E. Szabo.

[11] Martin Giese. First-order logic. In Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors, *Verification of Object-Oriented Software: The KeY Approach*, LNCS 4334, pages 21–65. Springer, 2006.

[12] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. *SIGPLAN Not.*, 43(6):281–292, 2008.

[13] Reiner Hähnle and Angela Wallenburg. Using a software testing technique to improve theorem proving. In Alexandre Petrenko and Andreas Ulrich, editors, *Post Conference Proceedings, 3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003), Montréal, Canada*, volume 2931 of *LNCS*, pages 30–41. Springer-Verlag, 2004.

[14] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, 2000.

[15] B. Hummel. *Generation of Induction Axioms and Generalizations*. PhD thesis, Universität Karlsruhe, 1990.

[16] Andrew Ireland and Alan Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming*, 9(2):225–245, 1999.

[17] Deepak Kapur. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity*, 19(3):307–330, 2006.

[18] Deepak Kapur and Mahadevan Subramaniam. Lemma discovery in automated induction. In *CADE-13: Proceedings of the 13th International Conference on Automated Deduction*, pages 538–552, London, UK, 1996. Springer-Verlag.

[19] Laura Kovács. Reasoning algebraically about p-solvable loops. In *TACAS*, pages 249–264. Springer, 2008.

[20] K. R. M. Leino and F. Logozzo. Using widenings to infer loop invariants inside an smt solver, or: A theorem prover as abstract domain. In *International Workshop on Invariant Generation (WING'07)*, pages 70–84, Hagenberg, Austria, June 2007. RISC.

[21] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *APLAS*, pages 119–134, 2005.

[22] Ola Olsson and Angela Wallenburg. Customised induction rules for proving correctness of imperative programs. In Bernhard Aichernig and Bernhard Beckert, editors, *Software Engineering and Formal Methods. 3rd IEEE International Conference, SEFM 2005, Koblenz, Germany, September 7–9, 2005, Proceedings*, pages 180–189. IEEE Computer Society Press, 2005.

[23] E. Rodríguez-Carbonell and D. Kapur. An Abstract Interpretation Approach for Automatic Generation of Polynomial Invariants. In *International Symposium on Static Analysis (SAS 2004)*, volume 3148 of *Lecture Notes in Computer Science*, pages 280–295. Springer-Verlag, 2004.

[24] E. Rodríguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *International Symposium on Symbolic and Algebraic Computation 2004 (ISSAC04)*, pages 266–273. ACM Press, 2004.

[25] Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.

[26] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using gröbner bases. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 318–329, New York, NY, USA, 2004. ACM.

[27] Angela Wallenburg. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNAI*, chapter Proving by Induction, pages 453–479. Springer, 2007.

[28] Angela Wallenburg. Generalisation of induction formulae based on proving by symbolic execution. Technical report, Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden, 2009.

[29] Christoph Walther. Mathematical induction. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming. Deduction Methodologies*, volume 2, chapter 3, pages 127–227. Oxford University Press, 1994.

[30] Christoph Walther, Markus Aderhold, and Andreas Schlosser. The L 1.0 primer. Technical Report VFR 06/01, FG Programmiermethodik, Technische Universität Darmstadt, 2006.

[31] Christoph Walther and Stephan Schweitzer. About VeriFun. In *Automated Deduction — CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 322–327. Springer-Verlag, 2003.

Appendix: Some Relevant Java Card DL Rules

$$\mathsf{allRight}\ \frac{\Gamma \Longrightarrow [x/c](\phi),\ \Delta}{\Gamma \Longrightarrow \forall x.\phi,\ \Delta}$$

with $c :\rightarrow A$ a new constant, if x:A

$$\mathsf{allLeft}\ \frac{\Gamma,\ \forall x.\phi, [x/t](\phi) \Longrightarrow \Delta}{\Gamma,\ \forall x.\phi \Longrightarrow \Delta}$$

with $t \in \mathrm{Trm}_{A'}$ ground, $A' \sqsubseteq A$, if x:A

$$\mathsf{impLeft}\ \frac{\Gamma \Longrightarrow \phi,\ \Delta \qquad \Gamma,\ \psi \Longrightarrow \Delta}{\Gamma,\ \phi \rightarrow \psi \Longrightarrow \Delta}$$

$$\mathsf{loopUnwind}\ \frac{\Longrightarrow \langle \pi\ \texttt{if (}e\texttt{) \{}p\ \texttt{while (}e\texttt{)}\ p\texttt{\}}\ \omega\rangle\phi}{\Longrightarrow \langle \pi\ \texttt{while (}e\texttt{)}\ p\ \omega\rangle\phi}$$