

Thesis for the Degree of Licentiate of Philosophy

# Induction Rules for Proving Correctness of Imperative Programs

Angela Wallenburg

**CHALMERS** | GÖTEBORG UNIVERSITY



Department of Computer Science and Engineering  
Chalmers University of Technology and Göteborg University  
SE-412 96 Göteborg  
Sweden

Göteborg, December 2004

Induction Rules for Proving Correctness of Imperative Programs  
Angela Wallenburg

© Angela Wallenburg, 2004

Technical Report no. 1L  
ISSN 1652-876X

Department of Computer Science and Engineering  
Chalmers University of Technology and Göteborg University  
SE-412 96 Göteborg, Sweden  
Telephone +46 (0)31-772 1000

Printed at Chalmers, Göteborg, 2004

## Abstract

This thesis is aimed at simplifying the user-interaction in semi-interactive theorem proving for imperative programs. More specifically, we describe the creation of *customised induction rules* that are tailor-made for the specific program to verify and thus make the resulting proof simpler. The concern is in user interaction, rather than in proof strength. To achieve this, two different verification techniques are used.

In the first approach, we develop an idea where a software testing technique, *partition analysis*, is used to compute a partition of the domain of the induction variable, based on the branch predicates in the program we wish to prove correct. Based on this partition we derive mechanically a partitioned induction rule, which then inherits the divide-and-conquer style of partition analysis, and (hopefully) is easier to use than the standard (Peano) induction rule.

The second part of the thesis continues with a more thorough development of the method. Here the connection to software testing is completely removed and the focus is on inductive theorem proving only. This time, we make use of failed proof attempts in a theorem prover to gain information about the problem structure and create the partition. Then, based on the partition we create an induction rule, in destructor style, that is customised to make the proving of the loop simpler.

With the customised induction rules, in comparison to standard (Peano) induction or Noetherian induction, the required user interaction is moved to an earlier point in the proof which also becomes more modularised. Moreover, by using destructor style induction we circumvent the problem of creating inverses of functions. The soundness of the customised induction rules created by the method is shown. Furthermore, the machinery of the theorem prover (KeY) is used to make the method automatic. The induction rules are developed to prove the total correctness of loops in an object-oriented language and we concentrate on integers.



# Contents

<b>Introduction</b>	<b>1</b>
1 Overview . . . . .	1
2 Motivation . . . . .	2
3 Our Infrastructure – a Semi-Interactive Theorem Prover . . . . .	3
4 Induction . . . . .	4
5 Using a Software Testing Idea – Partition Analysis . . . . .	5
6 Creating Customised Induction Rules from Partitions . . . . .	6
7 Using a Theorem Prover to Generate Partitions . . . . .	7
8 Destructor Style Induction . . . . .	8
9 More Related Work . . . . .	9
10 Summary and Contributions . . . . .	10
11 Future Work . . . . .	11
<b>Paper I: Using a Software Testing Technique to Improve Theorem Proving</b>	<b>17</b>
1 Introduction . . . . .	17
2 Motivating Example . . . . .	19
3 Computing Partitioned Induction Rules . . . . .	21
4 Examples . . . . .	23
4.1 Simple Example Revisited . . . . .	23
4.2 Russian Multiplication Example . . . . .	25
5 Limitations and Future Work . . . . .	27
<b>Paper II: Automatic Generation of Customised Induction Rules for Proving Correctness of Imperative Programs</b>	<b>31</b>
1 Introduction . . . . .	31
2 Motivation . . . . .	33
3 Preliminaries . . . . .	35
4 Induction . . . . .	37
5 Computing Customised Induction Rules . . . . .	38
5.1 Destructor Style Induction . . . . .	41
5.2 Branching . . . . .	42
6 Mechanizing the Approach Using a Theorem Prover . . . . .	43
6.1 Finding Predecessor Functions . . . . .	44
6.2 Extracting the Domain . . . . .	46

6.3	Finding Partitions . . . . .	46
7	Soundness . . . . .	46
8	A More Complex Example . . . . .	49
9	Comparison to Noetherian Induction . . . . .	50
10	Related Work . . . . .	51
11	Summary and Conclusions . . . . .	52
12	Future Work . . . . .	53

# Introduction

## 1 Overview

Software verification is the process of deciding to what degree a program meets its specification, that is the correctness of the program. It is increasingly demanded when lives and fortunes depend on the software. Two major classes of software verification techniques exist; software testing and formal verification.

*Software testing* is the process of executing the software repeatedly until either the system is exhaustively tested (real systems are too large for this to be remotely practical), the testing budget is used up, no new defects are found, or there is some other reason (usually probabilistic) to believe that the software is correct “enough”. *Formal verification* means reasoning about correctness in a way that is formal, or mathematics-based. Theorem proving is a mathematical process often used in formal software verification. It then generates either a mathematical proof or a failed proof attempt of the program correctness. The theorem proving process can be automated by computer programs to some extent depending on the underlying logic, however in general not to a sufficient level for most applications in software engineering. Thus, the user needs to guide the inferences made by the system. This can be done in interactive theorem provers. Also hybrid theorem provers exist, offering different levels of automation and interaction. The user interaction required is sometimes very complicated even for experts in formal methods.

In this thesis, we will look into both these verification techniques. First (in paper 1), we develop an idea where software testing is used, mainly as a source of inspiration, to simplify the user interaction in semi-interactive theorem proving. The second part of the thesis (paper 2) continues with a more thorough development of the method. Here the connection to software testing is completely removed and the focus is on inductive theorem proving only.

## 2 Motivation

To complete a proof of program correctness is often more difficult than writing the program itself. This is partly because there are fundamental difficulties in software verification which require advanced user interaction. One example is the reasoning about possibly unbounded objects, for instance integers, lists and other data types, and about recursively defined algorithms or program loops. In computer science, *mathematical induction* is the main method to reason about infinite structures.

The simplest and most common form of mathematical induction that can be used to prove that a statement holds for all natural numbers  $n$ , brings two proof obligations: (1) showing that the statement holds when  $n = 0$ , and (2) showing that if the statement holds for  $n = m$ , then the same statement also holds for  $n = m + 1$ . Along with the induction principle, proving these steps allows us to conclude that the statement holds for all natural numbers. This form of induction is called standard induction or Peano induction.

The following is a simple example of a loop that is not possible to prove totally correct using the standard induction rule (at least not without very complex user interaction). The description here is brief, details are explained in the following sections. Here is the (JAVA CARD) code of the loop:

```

final int c = ... ;
int i;
...
while(i > 0) {
  if(i >= c) {
    i = i - c;
  } else {
    i--;
  }
}

```

For this while-loop to terminate in a state where  $i = 0$  we need in the precondition that  $i \geq 0$  and  $c \geq 1$ .  $c$  is constant. In dynamic logic (the essentials of our logical framework are described later) the proof obligation then is  $\forall i \in \mathbb{N} \cdot \phi(i)$ , where  $\phi(i)$  is:

$$i \geq 0 \wedge c \geq 1 \rightarrow$$

$$\langle \text{while } (i > 0) \{$$

$$\quad \text{if } (i \geq c) \{$$

$$\quad \quad i = i - c;$$

$$\quad \text{ } \text{else } \{$$

$$\quad \quad i--;$$

$$\quad \text{ } \}$$

$$\} \rangle i = 0$$

The formula contains a total correctness assertion: the program within the diamond brackets  $\langle \rangle$  (here the code of the while-loop) terminates and in the final state the postcondition following the brackets must hold (here  $i = 0$ ).

Now, in an interactive theorem prover the user has to supply the induction variable and the formula to prove. In this example the obvious choice for the induction variable is  $i$  (see the terminating condition of the loop). Picking the right formula is generally a lot more complicated. The simplest possible choice for the induction hypothesis when proving correctness of a loop would be to take  $\phi(i)$ . It is completely schematic and requires no interaction with the user. This hypothesis, however, is too weak when using the standard induction rule. Roughly speaking, in a proof attempt of the standard step case,  $\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+1)$ , the following happens: the while-loop is unwound for  $n+1$  and the proof branches at the if-statement. One case (the one with “`i--;`”) is possible to prove, because “`(n+1)--;`” is equal to  $n$  after symbolic execution. The proof obligation for this case simplifies to  $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n < c \rightarrow \phi(n)$ , which is valid. In the other case symbolic execution gives  $n+1-c$  so that the resulting proof obligation  $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n \geq c \rightarrow \phi(n+1-c)$  is in general unprovable. With standard induction, a more powerful induction hypothesis must be found – a very difficult task for a user with no training in formal methods.

In this thesis we will develop customised induction rules to decrease and simplify user interaction in an interactive theorem prover, as well as a method for automatic construction of the rules. These induction rules are developed to be used proving total correctness of loops in an object-oriented language (JAVA CARD). We concentrate on natural numbers and integers.

### 3 Our Infrastructure – a Semi-Interactive Theorem Prover

In this work, the same theorem prover serves both as the subject of improvement as well as the implementation basis for automation of the method. It is a component of a larger verification system for which the target users are software engineers. With the help of the system users can create UML models, programs and constraints, and then formally analyze these. The system, called KeY [ABB<sup>+</sup>04], is integrated into a commercial CASE-tool and has been created with the outspoken aim of being a software-engineer-friendly tool for formal methods. As the programming language the system currently uses a single-threaded subset of JAVA, called JAVA CARD [Che00].

In the stand-alone theorem prover that we will focus on in this thesis, the verification paradigm is to execute programs with symbolic values, which then are checked (symbolically) against the formal specification. For a background reference to symbolic execution, see [HK76]. The logic used in the prover is JAVA CARD Dynamic Logic [HKT00], abbreviated DL. This DL has been extended specifically for the use of JAVA CARD, to a logic called JAVA CARD DL [Bec01, BS01a]. Dynamic logic is a first-order logic with modalities for partial,  $[p]$ , and total correctness,  $\langle p \rangle$ , where  $p$  is a JAVA CARD sequence. (More intricate modalities have been developed as well, see [BM03, BS01b].) For instance, the formula  $\phi \rightarrow \langle p \rangle \psi$  is valid if for every state satisfying the precondition  $\phi$ ,

a run of the program  $p$  starting in that state terminates (normally) in a state where the postcondition  $\psi$  holds. Dynamic logic allows the inclusion of program statements in the formulas for representation of states. Sequents in the calculus are written  $\phi_1, \dots, \phi_m \vdash \psi_1, \dots, \psi_n$  which has the same semantics as the formula  $\phi_1 \wedge \dots \wedge \phi_m \rightarrow \psi_1 \vee \dots \vee \psi_n$ . There are also facilities in the prover to deal with real `JAVA CARD` integers, but to keep presentation simple we use the mathematical integers instead. For details on the number system, see [BS04]. Worth noting, there is a mechanism in `JAVA CARD` Dynamic Logic to simplify the treatment of exceptions, aliasing, and other side effects, called *updates*, that we later make use of in the construction of induction rules. An update is of the form  $\{x := t\}$  where  $x$  is a program variable and  $t$  is a term and the intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e.  $\{x := t\}\phi$  has the same semantics as  $\langle \mathbf{x} = \mathbf{t}; \rangle \phi$ .

It is important to note that the method as such is not limited to any particular theorem prover.

## 4 Induction

Mathematical induction is an essential tool for reasoning about iterative and recursive structures and formulas. Also, it is the main obstacle in automating software verification to an acceptable degree. We have briefly seen the special case of induction over the natural numbers, also known as first order Peano-induction. Here is the rule in Dynamic Logic:

$$\frac{\Gamma \vdash \phi(0) \quad \Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i) \rightarrow \phi(i+1)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

A proof of the soundness of this rule can be found for example in the book [Hei95]. This form of induction allows us to prove a formula valid for all natural numbers. For instance, together with the precondition  $i \geq 0$ , the following formula is very well suited to be proved totally correct using Peano-induction:  $\langle \text{while}(i > 0) \ i--; \rangle i = 0$ . This is because it decrements by steps of one and ends at zero. However, if we alter the program slightly to for instance stepping down by division by 2, it becomes very hard to make immediate use of Peano-induction.

In most theorem provers it is possible to use the standard induction rule and “just” strengthen the induction hypothesis as soon as one encounters a non-trivial proof attempt. (This is the case for instance in KeY, the theorem prover that we use in this work.) However, this can be a formidable challenge even for formal methods experts. First, it is difficult to come up with a suitable generalisation of what one wants to prove. Second, in the remaining proof after the application of the induction rule the tiniest pieces of work, for example the instantiation of a quantified formula, require considering very large amounts of formal information.

A generalisation called Noetherian induction, or well-founded induction makes it easier to prove the same statement because the induction hypothesis is stronger. Using the well-founded induction principle, proving

$$\forall m \in M \cdot (\forall k \in M \cdot k \prec_M m \rightarrow \phi(k)) \rightarrow \phi(m)$$

and that  $(M, \prec_M)$  is a well-founded set, means that we have verified  $\forall m \in M \cdot \phi(m)$ . This is the most general form of mathematical induction. Using this rule, we would in many cases avoid the need for generalisation of the proof obligation, like in the example decrementing by division by 2. Still, the user-interaction involved in the remaining proof after application of the Noetherian induction rule can be very complicated because of the inherent mix of data and control-flow correctness and the size of the proof.

The customised induction rules created in this thesis, are in fact instances of well-founded induction, see paper 2. By using these customised induction rules, the gain is twofold; first, it decreases the need for generalisation and second, it branches the proof at an earlier stage so that the resulting proof becomes easier to develop or a failed proof attempt becomes easier to “debug”. The concern of this thesis is in user interaction, rather than in proof strength, and in paper 2 we also make a comparison between customised induction rules and Noetherian induction in this respect.

## 5 Using a Software Testing Idea – Partition Analysis

Following Dijkstra’s famous remark that testing can never prove the absence of errors in a computer program, only their presence [DDH72], was a long period of time when the two verification techniques were viewed as opposing ends in the spectrum of software quality assurance techniques. Still, testing is a core technique used by practitioners every day, while formal verification is difficult to master, and employed mostly by specialists in academia. Testing is only partially automated, and even less formalized. Although there have been industrial success stories in hardware verification, for software verification most practitioners agree that formal verification is too costly, cumbersome and difficult to be useful in practice [DR96]. However, during the last decade there has been an increasing interest into cross-pollinating ideas from the two fields, see for instance [PU04].

Most of these efforts to combine formal methods and software testing go in the direction of exploiting formal methods to solve testing problems. There are fairly established techniques for test case generation from formal specifications, for instance in B, Z [LPU02, ABC<sup>+</sup>02], VDM [Dro99], Haskell [CH00] and ASML [GGSV02]. The presence of a formal specification can also solve the oracle problem (that is the problem of deciding whether the observed outcome of program execution is acceptable or not). One obstacle of this approach is that availability of a formal specification is the exception rather than the rule [Kni98]. On the other hand, if the cost for providing a formal specification has

been invested already, one can use it as a basis not only for testing, but even for formal source code verification. Moreover, by employing techniques from testing, formal verification may actually come into reach. A particular technique from testing that can considerably simplify the verification effort, as shown in this thesis, is called *partition analysis*.

*Partition testing* is a software testing technique used to systematically reduce test volume. A program's possibly infinite input space is divided into a finite number of disjoint subdomains. Testing is done by picking one or more elements from each subdomain to form a test set that is somehow representative for the program behavior. Ideally, all elements in a subdomain behave in the same way with respect to the specification, that is, they are all processed correctly or they are all processed incorrectly. Subdomains with this property are called *revealing* [WO80] or *homogeneous* [HT90].

Interestingly, there is a line of work in software testing theory [GG75, How76, WO80, RC85, HT90], where it is shown that testing *can* be used to show the *absence* of errors *provided that* certain properties in the test case selection are fulfilled. In the context of partition testing, the sought-after property is that subdomains are revealing. Unfortunately, establishing this property in practice means usually to give a formal correctness proof (for each subdomain). Hence, given the difficulties of general theorem proving, this was often discarded as impractical. Our results may be considered as a step towards obtaining such correctness proofs practically, because it suggests that proving correctness for each subdomain separately requires less user interaction than giving a proof simultaneously for the entire domain (as usually done in theorem proving).

## 6 Creating Customised Induction Rules from Partitions

As shown, the use of induction is a major obstacle in automating software verification. The complexity of the induction, of course, depends on the complexity of the loop or method body and post condition at hand. In simple cases, the induction can be performed automatically. The key insight that we work out in this thesis is that the technique of partition testing is in fact a fairly general and automatic divide-and-conquer concept that can be used to simplify induction in formal verification proofs.

In short, to verify a loop, we use a white-box partition analysis based on the branch predicates of its body and condition, to compute a partition of the domain of the induction variable. For this we employ techniques readily available in the software testing community. Then we refine this partition to arrive at subdomains of a syntactic form that is suitable for generation of the new induction rule. Based on the refined partition, we derive a new (program-specific) induction rule with one base case for each finite subdomain of the partition and one step case for each infinite subdomain. The details of the method and the development of the following example can be found in paper 1.

Returning to the motivating example, and using the described approach we automatically create a new, partitioned induction rule, which has *two* base cases and one step case:

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \dots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+c) \tag{3}$$

These are constructed from a branch coverage partition of the induction variable  $i \in \mathbb{N}$ . For instance (2) above corresponds to the subdomain with all values of  $i$  causing the “else” branch inside the loop to be executed. Note that this partitioned induction rule is powerful enough to make the proof go through automatically with the unchanged induction hypothesis  $\phi(i)$  and that is just what we desire in our effort to minimise user interaction.

## 7 Using a Theorem Prover to Generate Partitions

We want to extend the approach described in the section above to overcome a few flaws in it. The most important one is that the partition analysis is based on the branch predicates. In fact, for the method to be successful (that means really providing a simplification) it is required that the branch predicates somehow capture what is being computed in the corresponding branch. This is of course not always the case. Consider the example:

```

i ≥ 0 ∧ c ≥ 1 →
  { while (i > 0) {
    if (i >= c) {
      i = i - (c + 1);
    } else {
      i--;
    }
  } } true

```

Here we have changed the statement that decrements the induction variable, for the case when  $i \geq c$ . The method now comes up with *the same* induction rule as before (regardless of the postcondition too), since it focuses on the branch predicates (in this branch  $i > 0 \wedge i \geq c$ ) to compute the partition. Unfortunately, this does not make induction simple at all (in the step  $i + c$  is not “cancelled out” by the  $i - (c + 1)$  statement). In the worst case, the branch predicates are completely unrelated to the induction variable, and we simply get no information from them on how to partition the domain of the induction variable.

Instead, we want to create the partition of the induction variable so that all elements within a subdomain undergo the same update by the loop body. This time, we decide to use the machinery of our theorem prover (KeY), to process

program-formulae by symbolic execution and attain the information (partition) we need. We simply apply the calculus rules to unwind the loop once and then apply all applicable calculus rules (except loop unwinding!) until no further progress can be made. This can be thought of as a naive attempt at proving the formula correct. Once the proof attempt is stuck, we will be left with an intact copy of the original loop, together with a set of updates and conditions. If there are branches inside the loop body these will result in several open proof branches after the proof attempt. We gather the terms that are not modal operators in the sequent and extract the conditions for each open branch. These conditions form the partition from which we can derive the induction rule tailored for the specific loop that is to be proved. For details and examples of this procedure, see paper 2.

## 8 Destructor Style Induction

Another problem using branch predicates as the basis for partitioning, is that the process of transforming general branch predicates into predicates of the form that the method requires is non-trivial; in particular for the process to be mechanized. It involves finding the inverse of functions. We circumvent this problem by using *destructor style induction*, also described in [Bun01]. In short, we replace the step case proof obligations of the form with successor functions,  $\forall i \in \mathbb{D}_s \cdot \phi(i) \rightarrow \phi(s(i))$ , with proof obligations of the form using predecessor functions,  $\forall i \in \mathbb{D}_s \cdot \phi(p(i)) \rightarrow \phi(i)$ . In this way we can use the machinery of the theorem prover and create rules directly from a failed proof attempt, as described in the previous section, without the hassle of inverting functions to successor style. For instance, consider the following example, squared steps:

$$\begin{aligned} &\forall i \in \mathbb{N} \cdot i > 1 \rightarrow \\ &\quad \langle \text{while } (i < 100) \{ \\ &\quad \quad i = i * i; \\ &\quad \} \rangle \text{true} \end{aligned}$$

Note the assignment “ $i = i * i$ ;” to the induction variable  $i$ . The idea is to perform induction as usual, but start “one step earlier”. Rather than using the successor style step  $\phi(i) \rightarrow \phi(\sqrt{i})$ , (where it would be non-trivial to derive the inverse of  $i^2$  to  $\sqrt{i}$  and its meaning in the context of our programming language) we directly use the update to the induction variable ( $i$ ), and get  $\phi(i^2) \rightarrow \phi(i)$ , and then the need for square roots is gone.

We have observed that when symbolically executing a loop body, then whatever it does to the induction variable is accumulated and recorded in an *update*. So we simply reuse a failed proof attempt as described and if we assume (or require) that there is an update involving the induction variable before we start the unwinding of the loop, then to construct the destructor function all we have to do after symbolic execution is to find the update to this induction variable.

## 9 More Related Work

In the first approach of this thesis, a combination of the software verification techniques testing and proving is used, and in the second part the work considers only semi-interactive theorem proving. Here we describe some related work in both these areas.

Apart from the work that has already been mentioned in section 5 there are further approaches to combine formal methods and software testing. For instance, there was an early effort [Gel78] to use test data to aid in proving program correctness. Similarly to the work in the first part of this thesis the method is able to deal with recursive functions, but was also unable to go beyond piecewise linear functions and their computation by elementary program constructs. In the field of dynamic analysis, we have to point out the technique to generate invariants inductively from test cases, to be found in [NE02]. In [DHT03] it is shown how the verification of functional programs can be aided by the execution of tests for sub-properties, thus failing proof attempts can be avoided. Note that in our work we actually do not perform any execution of programs (other than the symbolic execution in the JAVA CARD DL calculus), hence the connection to the testing field is rather weak. As stated earlier, we mainly use it as a source of inspiration.

Regarding the mechanization of induction proofs there is an overwhelming mass of work that has been going on for decades. A comprehensive description of this is out of the scope of this thesis, however we will give some pointers to relevant areas. First of all, there are two major approaches of mechanizing induction proofs – explicit and implicit induction. In *explicit* induction, proofs are built using an induction principle for which (explicit) rules are incorporated in the proof, for early seminal work see [Bur69, Aub79, BM79] and for good background references see [Wal94, Bun01, Sli97]. Using *implicit* induction [Red90, BKR95, KM87] one adds the conjecture to be proven to an equation system with existing axioms, then checks for inconsistencies and if none are found the conjecture is concluded to be an inductive theorem. Implicit induction allows a higher degree of automation. In this thesis we deal with explicit induction only, as we are concerned with computing induction rules for use in an interactive theorem prover. Whereas some of the mature techniques for explicit induction are quite powerful (having proved many difficult theorems [BM88, BvHSI90, BvHHS90]), they are often restricted to proving properties about programs written in tiny functional programming languages. In this thesis we merely scratch on the surface of automating induction proving, but we do so with the somewhat neglected concerns of a real imperative programming language and a shift of focus towards user-interaction.

Since the recurring goal in both parts of this work is to simplify the user-interaction in theorem proving for imperative programs, we here specify some other attempts at this task. For instance, in the B method [Abr96], loop correctness is proved using five different proof obligations, each with a distinct aim clear to the user. Interestingly, we have noticed that we also achieve some simplified user interaction using the customised induction rules, because the proof is split

at an early stage and separating the concerns of the different parts of the proof. However, we are relieved of the burden to annotate the program with an invariant and a variant, as in the B method (we might need to generalise our induction hypothesis, though). The reduction in user interaction that is achieved by separating the concerns of control and data correctness parts of proof goals has also been noticed in the hardware community [SKK94, HB95]. Other pieces of work to ease the user interaction in proving loop correctness are the automatic generation of loop invariants [RCK04a, RCK04b], interactive proof critics [IJR99] and rippling [BSvH<sup>+</sup>93]. Furthermore, ACL2 [BKM96, KM97] constitutes a big effort to create an industrial-strength user-guided automated theorem prover, however we find no particular concern for simplifying proofs by induction there.

## 10 Summary and Contributions

In this thesis we describe methods to automatically create customised induction rules for proving the total correctness of loops in an object-oriented language. The resulting rules are tailor-made for the respective loops to be verified. In comparison to standard (Peano) induction or Noetherian induction, the customised induction rules significantly simplify the user interaction required to perform the proof in a number of ways. The aim of the comparison is not to analyse the relative proof-strength but rather the usability and interaction requirements. For details of the comparison, see paper 2.

This work is a step towards automating the proving of loops in KeY. It has achieved a shift of focus for the user interacting with the prover. The method is not stronger than the well-founded induction, but the required interaction steps are moved to an earlier point in the proof and branching provides labels that make interaction more focused (if nothing else this makes writing tutorials easier). There are more branches but they are smaller and the user interaction required within each branch in the rest of the proof is simpler, for instance it is easier to find the correct instantiations. Another important consequence of this is that it is much easier for the user to analyse a failed proof attempt and see what kind of generalisation of the postcondition that might be necessary. Since the control flow has been lifted into separate proof branches we can now attack this problem with specialized methods. Here are some highlights of the work:

1. *Combined verification techniques:* we show (in paper 1) how a technique from the field of software testing can be used to improve theorem proving.
2. *Extension of the method;* we extend and improve our first approach (in paper 2) in the following way:
  - (a) *Use accurate information for partition* the new method uses the updates to the induction variable to derive induction steps, which improves the chances of the final customised induction rule to actually provide a simplification when applied.

- (b) *We expanded the range of loops the method can handle*, both by removing the limitation to natural numbers and being able to use loops iterating both up and down.
3. *We introduced the use of destructor style induction*, and in this way we avoid the issue of inverting the step functions, complicating the process.
4. *Separation of proof of control flow and data correctness was achieved*; this was realised through the introduction of separate proof obligations for well-foundedness (or termination). The separation means that it is clear to the user if the problem of the proof lies in the control flow (well-foundedness obligations) or data correctness (in the step cases). This improves the possibility to “debug” a failed proof attempt.
5. *The soundness of the customised induction rules was established*
6. *A way towards opening up a large variety of loops*, was also suggested, that would improve the usefulness of the method while being very simple to integrate.
7. *Showed how to use an automatic theorem prover to compute partition needed for the rules*. By using a theorem prover to create the partition needed for the rules, the connection to testing and the need for an external partition analysis tool are removed.

To summarise, one can see customised induction as a way of making well-founded induction more automatic, useful and deterministic, by extracting more information from the program.

## 11 Future Work

The most important future work involves extending the method to be used with other data structures than integers, proving partial correctness possibly using separate termination analysis, nested loops, multiple induction variables and the generalisation of postconditions. For a thorough listing of future work, refer to paper 2.

## References

- [ABB<sup>+</sup>04] 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 System Modeling*, 2004. Online First issue, to appear in print.

- [ABC<sup>+</sup>02] F. Ambert, F. Bouquet, S. Chemin, S. Guenaud, B. Legeard, F. Peureux, M. Utting, and N. Vacelet. BZ-Testing-Tools: a toolset for test generation from Z and B using constraint logic programming. In Rob Hierons and Thierry Jéron, editors, *Proc. FATES'02, Formal Approaches to Testing of Software, Workshop of CONCUR'02, Brno, Czech Republic*, pages 105–120. INRIA, Technical Report, August 2002.
- [Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.
- [Aub79] Raymond Aubin. Mechanizing structural induction. *Theor. Comput. Sci.*, 9:329–362, 1979.
- [Bec01] 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.
- [BKM96] Bishop Brock, Matt Kaufmann, and J. Strother Moore. Acl2 theorems about commercial microprocessors. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 275–293. Springer-Verlag, 1996.
- [BKR95] Adel Bouhoula, Emmanuel Kounalis, and Michael Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.
- [BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.
- [BM88] Robert Boyer and J. Strother Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.
- [BM03] Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card's transaction mechanism. In *Proceedings, Fundamental Approaches to Software Engineering (FASE), Warsaw, Poland*, LNCS 2621, pages 246–260. Springer, 2003.
- [BS01a] Bernhard Beckert and Bettina Sasse. Handling JAVA's abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. Technical Report DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.

- 
- [BS01b] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.
- [BS04] Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, LNCS. Springer, 2004. To appear.
- [BSvH<sup>+</sup>93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.
- [Bun01] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.
- [Bur69] R.M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12:41–48, 1969.
- [BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oysterclam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer, Berlin, Heidelberg, 1990.
- [BvHSI90] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer, Berlin, Heidelberg, 1990.
- [CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.
- [Che00] Zhiquan Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer’s Guide*. JAVA Series. Addison-Wesley, June 2000.
- [DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, London, 1972.
- [DHT03] Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. Verifying haskell programs by combining testing and proving. In *QSIC ’03: Proceedings of the Third International Conference on Quality Software*, page 272. IEEE Computer Society, 2003.

- 
- [DR96] David L. Dill and John Rushby. Acceptance of formal methods: Lessons from hardware design. *IEEE Computer*, 29(4):23–24, apr 1996.
- [Dro99] Georg Droschl. Design and Application of a Test Case Generator for VDM-SL. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 47–55, 1999.
- [Gel78] Matthew M. Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, May 1978.
- [GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, jun 1975.
- [GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 112–122. ACM Press, 2002.
- [HB95] Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 98–113. Springer-Verlag, 1995.
- [Hei95] James L. Hein. *Discrete structures, logic, and computability*. Jones and Bartlett Publishers, Inc., 1995.
- [HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3):331–353, 1976.
- [HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.
- [How76] William E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, sep 1976.
- [HT90] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, dec 1990.
- [IJR99] Andrew Ireland, Michael Jackson, and Gordon Reid. Interactive proof critics. *Formal Aspects of Computing*, 11(3):302–325, 1999.
- [KM87] Deepak Kapur and David R. Musser. Proof by consistency. *Artif. Intell.*, 31(2):125–157, 1987.

- [KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.
- [Kni98] John C. Knight. Challenges in the utilization of formal methods. In *Fault-Tolerant Systems: 5th International Symposium, FTRFTT'98*, pages 1–17, 1998.
- [LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods—Getting IT Right*, volume 2391 of *LNCS*, pages 21–40. Springer-Verlag, July 2002.
- [NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the international symposium on Software testing and analysis*, pages 229–239. ACM Press, 2002.
- [PU04] Alexandre Petrenko and Andreas Ulrich, editors. *Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003*, volume 2931 of *Lecture Notes in Computer Science*. Springer, 2004.
- [RC85] D.J. Richardson and L.A. Clarke. Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, 1985.
- [RCK04a] 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.
- [RCK04b] 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.
- [Red90] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 162–178. Springer, Berlin, Heidelberg, 1990.
- [SKK94] K. Schneider, T. Kropf, and R. Kumar. Control-path oriented verification of sequential generic circuits with control and data path. In *European Design and Test Conference (EDTC)*, pages 648–652, Paris, France, March 1994. IEEE Computer Society.

- [Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In Elsa L Gunter and Amy Felty, editors, *Proc. 10th International Theorem Proving in Higher Order Logics Conference*, volume 1275 of *LNCS*, pages 275–290. Springer-Verlag, 1997.
- [Wal94] 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.
- [WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, may 1980.

# Using a Software Testing Technique to Improve Theorem Proving

Reiner Hähnle and Angela Wallenburg

Chalmers University of Technology and Göteborg University  
Department of Computing Science  
SE-412 96 Göteborg, Sweden  
`{angelaw,reiner}@cs.chalmers.se`

## Abstract

Most efforts to combine formal methods and software testing go in the direction of exploiting formal methods to solve testing problems, most commonly test case generation. Here we take the reverse viewpoint and show how the technique of partition testing can be used to improve a formal proof technique (induction for correctness of loops). We first compute a partition of the domain of the induction variable, based on the branch predicates in the program code of the loop we wish to prove. Based on this partition we derive mechanically a partitioned induction rule, which is (hopefully) easier to use than the standard induction rule. In particular, with an induction rule that is tailored to the program to be verified, less user interaction can be expected to be required in the proof. We demonstrate with a number of examples the effectiveness of our method.

## 1 Introduction

Testing and formal verification at first glance seem to be at opposing ends in the spectrum of techniques for software quality assurance. Testing is a core technique used by practitioners every day, while formal verification is difficult to master, and employed by specialists in academia. Testing is only partially automated, and even less formalized. Most practitioners agree that formal verification is too cumbersome and difficult to be useful in practice. On the other hand, testing cannot be used on its own to prove the absence of errors, because exhaustive testing is usually impossible. In practice, one stops testing once the number of found errors drops below a certain threshold (or simply when the testing budget is used up). Formal verification, although costly, can ensure that a program meets its (formal) specification for any input. Given this state of affairs, it might seem surprising that testing and verification can fruitfully interact – nevertheless, this is what we want to show in the present paper.

There is one fairly established connection between formal methods and testing (documented, for example, in several papers collected in this proceedings): test case generation from formal specifications. The presence of a formal specification can also solve the oracle problem. One obstacle of this approach is that availability of a formal specification is the exception rather than the rule. On the other hand, if the cost for providing a formal specification has been invested already, one can use it as a basis not only for testing, but even for formal source code verification. The contribution of this paper is to show that techniques from testing can considerably simplify the verification effort. Hence, the availability of a formal specification is doubly useful: on the one hand, with by now established techniques one can generate test cases automatically. In addition, as we show below, by employing techniques from testing, even formal verification may come into reach. We see our work as a first step towards a framework, where both testing and verification can be usefully combined.

*Partition testing* is a software testing technique used to systematically reduce test volume. A program's possibly infinite input space is divided into a finite number of disjoint subdomains. Testing is done by picking one or more elements from each subdomain to form a test set that is somehow representative for the program behaviour. Ideally, all elements in a subdomain behave in the same way with respect to the specification, that is, they are all processed correctly or they are all processed incorrectly. Subdomains with this property are called *revealing* [WO80] or *homogeneous* [HT90].

There is a line of work in software testing theory [GG75, How76, WO80, RC85, HT90], where it is shown that testing *can* be used to show the *absence* of errors *provided that* certain properties in the test case selection are fulfilled. In the context of partition testing, the sought-after property is that subdomains are revealing. Unfortunately, establishing this property in practice means usually to give a formal correctness proof (for each subdomain). Hence, given the difficulties of general theorem proving, this was often discarded as impractical. Our results may be considered as a step towards obtaining such correctness proofs practically, because it suggests that proving correctness for each subdomain separately requires less user interaction than giving a proof simultaneously for the entire domain (as usually done in theorem proving).

In a nutshell, here is what we do: the implementation basis for our work is a software verification system for the programming language JAVA CARD called KeY [ABB<sup>+</sup>02]. The verification paradigm of KeY is to execute programs with symbolic values, which then are checked (symbolically) against the formal specification. For a background reference to symbolic execution, see [HK76].

The main obstacle in automating software verification to an acceptable degree is the handling of programs with loops or recursive methods. These constructs require induction on one of the inductive data structures occurring in the program (for example, numbers or lists). The difficulty is to find a suitable induction hypothesis. This can be a formidable challenge even for formal methods experts. The complexity of the induction, of course, depends on the complexity of the loop or method body and post condition at hand. In simple cases, the induction can be performed automatically. Therefore, it would be extremely

beneficial to simplify the required induction hypotheses. The key insight that we work out in the present paper is that the technique of partition testing is in fact a fairly general and *automatic* divide-and-conquer concept that can be used to simplify inductions in formal verification proofs.

Roughly speaking, to verify a loop, we use a white-box partition analysis based on the branch predicates of its body and condition, to compute a partition of the domain of the induction variable. This partition is then used to derive (mechanically) an induction rule which takes the partition into account: let us call the *standard induction rule* for natural numbers the rule that allows to conclude that a statement  $\phi(n)$  holds for all  $n \in \mathbb{N}$  provided that it holds for the single base case (“ $\phi(0)$ ”) and for the single step case (“for any  $i$ , if  $\phi(i)$  then  $\phi(i + 1)$ ”). This is replaced by an induction rule that has  $m$  base cases and  $r$  step cases, each of which matches a subdomain of the partition and, hopefully, needs much less user interaction.

Other work that is related to this can be found elsewhere. For instance, there was an early effort [Gel78] to use test data to aid in proving program correctness. Recently, a technique to generate invariants inductively from test cases [NE02] has been presented. Furthermore, in [Sli97] it is described how to formally derive induction schemes for recursively defined functions.

The remainder of the paper is organized as follows. We start with a motivating example in Sect. 2. In Sect. 3 we describe the method. Then we show it at work. First, we revisit the introductory example (Sect. 4.1), followed by a more sophisticated problem (Sect. 4.2). We close by pointing out current limitations (and, hence, future work).

## 2 Motivating Example

In this section we describe a simple example of a loop that is not possible to prove (without complex user interaction) using a standard induction rule, but is easy with our approach. The description here is brief. Our method is explained in detailed in the following section. Here is the JAVA CARD code of the loop:

```
int final c = ... ;
int i;
...
while(i > 0) {
  if(i >= c) {
    i = i - c;
  } else {
    i--;
  }
}
```

For this while-loop to terminate in a state where  $i = 0$  we need in the precondition that  $i \geq 0$  and  $c \geq 1$ .  $c$  is constant. In dynamic logic (briefly DL – the essentials of our logical framework are described in Sect. 4.1) the proof

obligation is  $\forall i \cdot \phi(i)$ , where  $\phi(i)$  is:

$$\begin{aligned}
 & i \geq 0 \wedge c \geq 1 \rightarrow \\
 & \langle \text{while } (i > 0) \{ \\
 & \quad \text{if } (i \geq c) \{ \\
 & \quad \quad i = i - c; \\
 & \quad \} \text{ else } \{ \\
 & \quad \quad i--; \\
 & \quad \} \\
 & \rangle i = 0
 \end{aligned}$$

The formula contains a total correctness assertion: the program within the brackets  $\langle \rangle$  (here the code of the while-loop) terminates and in the final state the postcondition following the brackets must hold (here  $i = 0$ ).

The simplest possible choice for the induction hypothesis when proving correctness of the loop is to take  $\phi(i)$ . It is completely schematic and requires no interaction with the user. This hypothesis, however, is too weak when using the standard induction rule. Roughly speaking, in a proof attempt of the standard step case,  $\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+1)$ , the following happens: the while-loop is unwound for  $n+1$  and the proof branches at the if-statement. One case (the one with “ $i--$ ;”) is possible to prove, because “ $(n+1)--$ ;” is equal to  $n$  after symbolic execution. The proof obligation for this case simplifies to  $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n < c \rightarrow \phi(n)$ , which is valid. In the other case symbolic execution gives  $n+1-c$  so that the resulting proof obligation  $\forall n \in \mathbb{N} \cdot \phi(n) \wedge n \geq c \rightarrow \phi(n+1-c)$  is in general unprovable. With standard induction, a more powerful induction hypothesis must to be found – a difficult task for a user with no training in formal methods!

In our approach we instead *automatically* create a new, partitioned induction rule. For our example loop the partitioned induction rule has *two* base cases and one step case:

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \dots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+c) \tag{3}$$

These are constructed from a branch coverage partition of the induction variable  $i$ . For instance (2) above corresponds to the subdomain with all values of  $i$  causing the “**else**” branch inside the loop to be executed. The creation of the partitioned induction rule for this particular example is described in more detail in Sect. 4.1. Note that this partitioned induction rule is powerful enough to make the proof go through automatically with the unchanged induction hypothesis  $\phi(n)$  that is just what we desire in our effort to minimise the user interaction.

### 3 Computing Partitioned Induction Rules

The idea is to create for each loop that we want to prove a new, tailor-made induction rule based on partitions. A partition is used, through the new induction rule, to divide the proof into smaller and hopefully simpler (in terms of user interaction) parts. Here is an overview of the method:

1. Compute a partition based on the branch predicates in the program code. We employ techniques readily available in the software testing community. Details are out of the scope of this paper, but the approach we use here is similar to the construction of the implementation partition in [RC85].
2. Refine this partition, thereby making use of the implicit case distinction contained in operators (such as `mod` or `÷`) that occur in branch predicates. The goal of the partition refinement is to arrive at subdomains of a syntactic form that is suitable for generation of the new induction rule.
3. Based on the refined partition, create a new (program-specific) induction rule with one base case for each finite subdomain of the partition and one step case for each infinite subdomain.
4. Prove correctness of the loop as usual, but use the new induction rule. This requires typically less user interaction than with the standard induction rule.

Now to the details. Specifically, assume that we have a program loop with input domain, or in our case, a domain of the variable that we want to perform induction over:  $D \subseteq \mathbb{N}$ . From a partition analysis as described in step 1 above we obtain a finite number of disjoint subdomains, say,  $D = D_1 \cup \dots \cup D_m$ . Let  $d_i$  be the characteristic predicate for each  $i \in 1, \dots, m$  with  $x \in D_i$  iff  $d_i(x)$  holds. Hence,  $x \in D$  iff  $d_1(x) \vee \dots \vee d_m(x)$ . The  $d_i$  are called *branch predicates*.

The branch predicates originate from the branching conditions in the program code and might contain operators defined by case distinction, for instance, `÷`, `mod`, and `≥`. These implicit case distinctions drive further partitioning.

For each such operator, if necessary, we create a partition such that each case distinction in the definition of the operator gives rise to a new subdomain. In the future, we plan to create a library of partitions for all operators that occur in `JAVA CARDexpressions` and the standard `JAVA CARDAPI` so that refining partitions can be looked up mechanically. In general, we strive to refine the original partition to obtain new subdomains of a particular *syntactic form*:

1.  $\{\}$  (the empty set)
2. A finite set  $\{x_1, \dots, x_k\}$ . Such a set is important to distinguish because it can quite simply be used as a base case in the new induction rule.
3. An infinite set of the form  $\{\lambda x.f(x) \mid x \in C\}$ , where  $C \subseteq \mathbb{N}$ . It is important that  $f(x)$  always increases its argument, because it is eventually to be used as an induction step in our new induction rule. We use  $\lambda x.f(x)$  because

we aim for an *expression* (and not the value of a function) to describe a set of values that we want to perform induction over.

Here is an example of a partition refinement based on an operator definition by case distinction: in the example from Sect. 2 one of the branch conditions contains the operator  $\geq$ , which has an implicit case distinction. We use the following definition of  $\geq$ :

$$x \geq z = \begin{cases} \text{true} & \text{if } \exists y \in \mathbb{N} \cdot (x = z + y) \\ \text{false} & \text{if } \exists y \in \mathbb{N} \cdot (x = z - 1 - y) \end{cases}$$

Each case gives directly a simple expression of the desired form:  $\lambda n.(c + n)$ , respectively,  $\lambda n.(c - 1 - n)$  would be used to refine a subdomain defined by the predicate  $i \geq c$ .

More precisely, we refine the subdomain of the original partition ( $\{i \in \mathbb{N} \mid i \geq c\}$ ) into two new subdomains by replacing  $i$  with  $(c + n)$  and  $(c - 1 - n)$  respectively. We then get  $\{c - 1 - n \mid n \in \mathbb{N} \wedge c - 1 - n \geq c\} = \{\}$  and  $\{c + n \mid n \in \mathbb{N} \wedge c + n \geq c\} = \{c + n \mid n \in \mathbb{N}\}$ , which are both of the form required above. The latter can be used to derive an induction step case.

Assume now that we have a refined partition of the syntactic form detailed above, where operators with implicit case distinctions are eliminated.

We create a new induction rule with the following set of proof obligations:

1. For each non-empty finite subdomain  $\{x_1, \dots, x_k\}$ , we create a base case consisting of the proof obligation

$$\phi(x_1) \wedge \dots \wedge \phi(x_k)$$

2. For each infinite subdomain  $D_i$  a new step case needs to be proven:

$$\forall n \in C_i \cdot \phi(n) \rightarrow \phi((\lambda x.f_i(x))n)$$

For the new induction rule to be sound it is important that some criteria are fulfilled:

1. For each step case of the form  $\forall n \in C_i \cdot \phi(n) \rightarrow \phi((\lambda x.f_i(x))n)$  the following holds:

$$\forall n \in C_i \cdot f_i(n) > n$$

This is to ensure that it really is a step, and it is achieved by constructing  $f_i(x)$  such that it increases its argument.

2. Each element of the domain  $D$  of the induction variable is covered in at least one of the step or base cases. Let  $B$  be the union of all finite subdomains giving rise to a base case and let  $f_1, \dots, f_r$  be the functions that define the step cases. Then we require

$$\forall x \in D \cdot (\exists k \in \{1, \dots, r\} \cdot \exists y \in C_k \cdot x = f_k(y)) \vee x \in B$$

This property is guaranteed by construction, because the partition property is invariant in the process; we only refine partitions or do not change them at all.

The first property entails that the minimal element of  $D$  cannot be in the subdomain defined by any step case. The second property says that all elements of  $D$  is in either a step case or a base case. As a consequence, there must be at least one base case of the induction.

## 4 Examples

### 4.1 Simple Example Revisited

Now we return to the motivating example from Sect. 2 and show how we actually computed the partitioned induction rule for it.

In KeY, the logical infrastructure is JAVA CARD DL [Bec01], an extension of dynamic logic (DL) [HKT00] to handle side effects, aliasing, exceptions and other complications of a real object-oriented programming language such as JAVA CARD. In DL, a formula  $\varphi \rightarrow \langle p \rangle \psi$  is valid if for every state  $s$  satisfying precondition  $\varphi$  a run of the program  $p$  starting in  $s$  terminates, and in the terminating state the post-condition  $\psi$  holds. The proof obligation from Sect. 2, that was written using pure DL, is slightly more complicated in JAVA CARD DL. Our proof obligation is  $\forall i_l \cdot \phi(i_l)$ . Let  $\phi(i_l)$  be the following formula:

$$\begin{aligned}
 & i_l \geq 0 \wedge c_l \geq 1 \rightarrow \\
 & \{i := i_l\} \{c := c_l\} \langle \text{while } (i > 0) \{ \\
 & \quad \text{if } (i \geq c) \{ \\
 & \quad \quad i = i - c; \\
 & \quad \} \text{else } \{ \\
 & \quad \quad i--; \\
 & \quad \} \\
 & \} \rangle i = 0
 \end{aligned}$$

The curly brackets in front of the formula are called (state) *updates*. Updates are the JAVA CARD DL solution to deal with aliasing and assignment in the calculus. They are basically primitive assignments of the form  $\{loc := val\}$  where  $val$  must be a logical (side effect free) term and  $loc$  a program variable.

To prove correctness of the loop we need to perform induction on the variable  $i$ . In JAVA CARD DL one cannot quantify over program variables, so the induction variable is a corresponding logical variable  $i_l$ . The domain of the induction variable is  $\mathbb{N}$ . From the branching conditions in the program we obtain the first partition of  $i$ 's domain:

$$\begin{aligned}
 D_1 &= \{x \in \mathbb{N} \mid d_1(x)\} = \{x \in \mathbb{N} \mid x \leq 0\} = \\
 &= \{0\} \\
 D_2 &= \{x \in \mathbb{N} \mid d_2(x)\} = \{x \in \mathbb{N} \mid x > 0 \wedge x < c\} = \\
 &= \{1, \dots, c-1\} \\
 D_3 &= \{x \in \mathbb{N} \mid d_3(x)\} = \{x \in \mathbb{N} \mid x > 0 \wedge x \geq c\} = \\
 &= \{x \in \mathbb{N} \mid x \geq c\}
 \end{aligned}$$

The subdomains  $D_1 = \{0\}$  and  $D_2 = \{1, \dots, c-1\}$  are finite and thus already in one of our desired formats.

Then, to refine/rewrite the original subdomain  $D_3$ , remember from Sect. 3 that for the operator  $x \geq z$ , we may use the expressions  $\lambda y.z+y$  and  $\lambda y.z-1-y$  to refine a subdomain. This gives a refinement of  $D_3 = \{x \in \mathbb{N} \mid x \geq c\}$  into two new subdomains  $D_3 = D_{31} \cup D_{32}$ , where

$$\begin{aligned} D_{31} &= (\text{replace } x \text{ in } D_3 \text{ with } c+y) \\ &= \{c+y \mid y \in \mathbb{N} \wedge c+y \geq c\} \\ &= \{c+y \mid y \in \mathbb{N}\} \\ D_{32} &= (\text{replace } x \text{ in } D_3 \text{ with } c-1-y) \\ &= \{c-1-y \mid y \in \mathbb{N} \wedge c-1-y \geq c\} \\ &= \{\} \end{aligned}$$

So the new subdomains  $D_1$ ,  $D_2$  and  $D_{31}$  are of the form we need to construct the new induction rule. To prove  $\forall n \in \mathbb{N} \cdot \phi(n)$ , it is then enough to prove

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \dots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(c+n) \tag{3}$$

where (1) is a base case and covers  $D_1$ , (2) is also a base case and covers  $D_2$ , and (3) is a step case that covers all elements in the subdomain  $D_{31}$ .

The proving process in KeY is partially automated, though it is an interactive theorem prover. When using the partitioned induction rule above, the following kinds of user interaction are required to complete the proof:

*Instantiation* means a single quantifier elimination by supplying a suitable instance term. In the KeY system, the user can simply drag-and-drop the desired term.

*Induction rule application:* when applying the partitioned induction rule, one can state the induction hypothesis by drag-and-dropping the existing proof obligation, and then pick the induction variable.

*Unwinding* of the loop needs to be initiated, but is done automatically.

*Decision procedure* is an automatic procedure that tries to decide the validity of arithmetic expressions over the integers. The decision procedure is sound but not complete. The user decides when (if) to run it.

Compared to the user interaction needed when using standard induction, this is less complicated. Using standard induction, if one uses the unmodified induction hypothesis and the same induction variable as above, one is left with an open proof goal and no rules to apply: one has to figure out a strong enough induction hypothesis.

## 4.2 Russian Multiplication Example

Let us see how the method works for proving the correctness of a more complicated algorithm – russian multiplication. The loop has more complicated control flow than in the previous example.

```
int russianMultiplication(int a,int b) {
  int z = 0;
  while (a != 0) {
    if (a mod 2 != 0) {
      z = z + b;
    }
    a = a/2;
    b = b*2;
  }
  return z;
}
```

For this loop we have the precondition  $a_0 \geq 0$  and the post-condition  $z = z_0 + a_0 * b_0$ , where  $a_0, b_0, z_0$  are the values of  $a, b$  and  $z$  before the loop. In JAVA CARD DL the proof obligation for the total correctness of this loop is  $\forall a_0 \cdot \phi(a_0)$ , where  $\phi(a_0)$  is

$$\forall b_0 \cdot \forall z_0 \cdot a_0 \geq 0 \rightarrow \{a := a_0\}\{b := b_0\}\{z := z_0\} \langle \text{while } (a \neq 0) \{ \text{if } (a \bmod 2 \neq 0) \{ z = z + b; \} a = a / 2; b = b * 2; \} \rangle z = z_0 + a_0 * b_0$$

where  $a_0, b_0$  and  $z_0$  are new logical variables.

This cannot be proven using standard induction unless the induction hypothesis is strengthened in a non-trivial way. In an attempt to prove the standard step case using  $\phi(a)$  as the induction hypothesis in  $\forall a \in \mathbb{N} \cdot \phi(a) \rightarrow \phi(a+1)$ , after unwinding and symbolically executing the loop we end up with  $\forall a \in \mathbb{N} \cdot \phi(a) \rightarrow \phi((a+1)/2)$  which is unprovable without induction.

Now let us compute a partitioned induction rule for this loop instead. The induction variable is  $a$  (the corresponding logical variable is  $a_0$ ). Its domain is  $\mathbb{N}$  and the first partitioning, using branch predicates, gives the following subdomains:

$$\begin{aligned} D_1 &= \{x \in \mathbb{N} \mid d_1(x)\} = \{x \in \mathbb{N} \mid x = 0\} \\ &= \{0\} \\ D_2 &= \{x \in \mathbb{N} \mid d_2(x)\} \\ &= \{x \in \mathbb{N} \mid x \neq 0 \wedge x \bmod 2 \neq 0\} \\ D_3 &= \{x \in \mathbb{N} \mid d_3(x)\} \\ &= \{x \in \mathbb{N} \mid x \neq 0 \wedge x \bmod 2 = 0\} \end{aligned}$$

In words, we have the singleton set containing zero and the sets with the odd and (non-zero) even numbers respectively.

Consider the branch predicate  $d_3(x) \leftrightarrow x \neq 0 \wedge x \bmod 2 = 0$ , which defines subdomain  $D_3$ . The definition of  $d_3(x)$  contains an operator with implicit case distinction: `mod`. We look up the definition of `mod 2`:

$$x \bmod 2 = \begin{cases} 0 & \text{if } \exists y \in \mathbb{N} \cdot (x = 2 * y) \\ 1 & \text{if } \exists y \in \mathbb{N} \cdot (x = 2 * y + 1) \end{cases}$$

Hence, we use the expressions  $\lambda y. 2 * y$  and  $\lambda y. 2 * y + 1$  to refine the original partition. Using the case distinction in the definition of  $x \bmod 2$ , gives us the refinement of the original subdomain  $D_3 = \{x \in \mathbb{N} \mid x \neq 0 \wedge x \bmod 2 = 0\}$  into two new subdomains  $D_3 = D_{31} \cup D_{32}$ :

$$\begin{aligned} D_{31} &= (\text{replace } x \text{ in } D_3 \text{ with } 2 * y) \\ &= \{2 * y \mid y \in \mathbb{N} \wedge (2 * y) \neq 0 \wedge (2 * y) \bmod 2 = 0\} \\ &= \{2 * y \mid y \in \mathbb{N} \wedge y \neq 0 \wedge 0 = 0\} \\ &= \{2 * y \mid y \in \mathbb{N}_1\} \\ D_{32} &= (\text{replace } x \text{ in } D_3 \text{ with } 2 * y + 1) \\ &= \{2 * y + 1 \mid y \in \mathbb{N} \wedge 2 * y + 1 \neq 0 \wedge (2 * y + 1) \bmod 2 = 0\} \\ &= \{2 * y + 1 \mid y \in \mathbb{N} \wedge 1 = 0\} \\ &= \{\} \end{aligned}$$

Similarly, for the branch predicate  $d_2(x)$  of the original partition, we get

$$\begin{aligned} D_{21} &= \{2 * y + 1 \mid y \in \mathbb{N}\} \\ D_{22} &= \{\} \end{aligned}$$

After refinement, we have non-empty subdomains of the form:

$$\begin{aligned} D_1 &= \{0\} \\ D_{21} &= \{2 * y + 1 \mid y \in \mathbb{N}\} \\ D_{31} &= \{2 * y \mid y \in \mathbb{N}_1\} \end{aligned}$$

Thus, the new subdomains have the syntactic form we need to construct an induction rule. With this rule, to prove  $\forall n \in \mathbb{N} \cdot \phi(n)$ , it is enough to prove that

$$\phi(0) \tag{1}$$

$$\forall n \in \mathbb{N}_1 \cdot \phi(n) \rightarrow \phi(2 * n) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(2 * n + 1) \tag{3}$$

where (1) is the base case, covering  $D_1$ , (2) the first step case, covering all elements in the subdomain  $D_{31}$  and (3) the second step case, covering  $D_{21}$ .

To prove the russian multiplication algorithm in KeY with the partitioned induction rule the required user interactions are basically the same as in the previous example. In particular, the induction now goes through completely unmodified.

## 5 Limitations and Future Work

In this paper we demonstrated that the technique of partition testing can be turned into a divide-and-conquer concept to simplify inductions in formal verification proofs. We have defined a syntactic framework that allows us to derive tailor-made induction rules based on partitions *automatically* and *mechanically*. Resulting induction rules are sound and complete by construction. Several examples were carried out not just by hand, but as concrete experiments in an interactive theorem prover. The experimental findings confirmed our conjecture. We think that our work is a first step towards a framework, where both testing and formal verification can be usefully combined.

In the current setting, our method has a number of limitations. Furthermore, its reach could be extended considerably. For a start, we considered induction not over arbitrary inductive data structures, but only the natural numbers. Future work is to extend our approach to also include induction over lists, trees, etc.

Our focus has been entirely on the verification of loops, and not arbitrary programs. Since loops are usually the major source of complexity in verification, in testing as well as in theorem proving, it is here that we expect the largest gain. Still, we also wish to investigate the idea of partitioning proofs for loop-free programs.

Clearly, not all induction proofs can be simplified with our approach. The crucial point is that our method requires that the branch predicates somehow capture what is being computed in the corresponding branch. This is often the case, but not always. If the branch predicates are completely unrelated to the induction variable, we simply get no information from the branch predicates on how to partition the domain of the induction variable. For instance in array-sorting algorithms, it is common that the induction goes over the indexes of the array, but the branch predicates typically have a comparison between the elements to be sorted and these might be randomly ordered. In future work we plan to remedy this by also using the weakest preconditions of updates to induction variables when we refine the partition. In KeY there is already a strongest postcondition generator.

Finally, the process of transforming general branch predicates into predicates of the form that our method requires (using  $\lambda$ -expressions) is non-trivial; in particular for the process to be mechanised. In our examples, quite simple branch conditions occurred. Future work includes dealing with predicates containing arbitrary linear operators and method calls, but quadratic operators and operators like  $\sin$ , we expect to be beyond reach. However, our method is conservative in the sense that if it does not find a useful refinement of a partition for a certain subdomain, the subdomain stays the same. In that case the proof will not be simplified, but it will not be more complicated either.

## Acknowledgements

We thank the anonymous reviewers and the editors for their many helpful comments. This work was also supported by a STINT (the Swedish Foundation for International Cooperation in Research and Higher Education) grant.

## References

- [ABB<sup>+</sup>02] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, and Peter H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *LNCS*, pages 327–330. Springer-Verlag, 2002.
- [Bec01] 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, Cannes, France*, volume 2041 of *LNCS*, pages 6–24. Springer-Verlag, 2001.
- [Gel78] Matthew M. Geller. Test data as an aid in proving program correctness. *Communications of the ACM*, 21(5):368–375, May 1978.
- [GG75] John B. Goodenough and Susan L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, jun 1975.
- [HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3):331–353, 1976.
- [HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.
- [How76] William E. Howden. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, sep 1976.
- [HT90] Dick Hamlet and Ross Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, dec 1990.
- [NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proceedings of the international symposium on Software testing and analysis*, pages 229–239. ACM Press, 2002.

- 
- [RC85] D.J. Richardson and L.A. Clarke. Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, 1985.
- [Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In Elsa L Gunter and Amy Felty, editors, *Proc. 10th International Theorem Proving in Higher Order Logics Conference*, volume 1275 of *LNCS*, pages 275–290. Springer-Verlag, 1997.
- [WO80] Elaine J. Weyuker and Thomas J. Ostrand. Theories of Program Testing and the Application of Revealing Subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, may 1980.



# Automatic Generation of Customised Induction Rules for Proving Correctness of Imperative Programs

Ola Olsson and Angela Wallenburg

Chalmers University of Technology and Göteborg University  
Department of Computing Science  
SE-412 96 Göteborg, Sweden  
`angelaw@cs.chalmers.se`

## Abstract

In this paper we develop a method for automatic construction of customised induction rules for use in a semi-interactive theorem prover. The induction rules are developed to prove the total correctness of loops in an object-oriented language. We concentrate on integers. First we compute a partition of the domain of the induction variable. Our method makes use of failed proof attempts in the theorem prover to gain information about the problem structure and create the partition. Then, based on this partition we create an induction rule, in destructor style, that is customised to make the proving of the loop simpler. Our concern is in user interaction, rather than in proof strength.

Using the customised induction rules, some separation of proof of control flow and data correctness is achieved, and we find that in comparison to standard (Peano) induction or Noetherian induction, simpler user interaction can be expected. Furthermore, by using destructor style induction we circumvent the problem of creating inverses of functions. We show the soundness of the customised induction rules created by the method. Furthermore, we use the machinery of the theorem prover (KeY) to make the method automatic. Several interesting areas are also identified that could open up for a larger range of loops the method can handle, as well as pointing towards full automation of these cases.

## 1 Introduction

A number of interactive theorem provers such as KeY [ABB<sup>+</sup>04] (see section 10 for more examples), have been around for a while. An industrial breakthrough of such tools is prevented by the large threshold for software engineers to use them. This is in part because of poor user interfaces and lack of training in

formal methods. However, there are also fundamental difficulties in software verification which naturally require advanced interaction.

One example is the reasoning about possibly unbounded objects, for instance integers, lists and other data types, and about recursively defined algorithms or program loops. In computer science, induction is the main method to reason about infinite (and finite) structures. In most theorem provers it is possible to use standard (Peano) induction to prove properties about natural numbers and “just” strengthen the induction hypothesis as soon as one encounters a non-trivial proof attempt. However, the user interaction associated with this can be immensely complicated. First, the induction rule at hand might not be strong enough to prove the original proof obligation. Second, it is difficult to come up with a suitable generalisation. Third, in the remaining proof after the application of the induction rule the tiniest pieces of work, for example the instantiation of a quantified formula, require considering large amounts of formal information.

There are earlier efforts (and successes in) automating induction proving; see [Bun01] for an overview. Most of this work is made for small functional programming languages. Part of the challenge that we have here however, derives from the circumstance that we work with an imperative programming language.

In this paper we present a method to automatically discover customised induction rules ultimately generating simpler inductive proofs. The main thrust is to make use of information from the program code to be able to devise better induction steps. The starting point is the idea presented in the paper [HW04]<sup>1</sup> where the domain of the induction variable is partitioned into subdomains and an induction step created for each of them. The particular approach relied on an external tool performing the partition analysis, based on the branch predicates of the program to prove. Here we extend that work by using a different source for partitioning, letting a theorem prover itself do that job and adding destructor style to the resulting rules. Our goal is to simplify user interaction when proving loop correctness. Furthermore, when user interaction has been simplified enough, there is none at all, and automatic proving is of course the ultimate goal for such a tool.

The same theorem prover serves both as our infrastructure as well as our subject of improvement. The theorem prover is a component of the KeY [ABB<sup>+</sup>04] system, a software verification system for which the target users are software engineers. The basis for the deduction component is a first order Dynamic Logic [HKT00]. More precisely we use a version of the logic extended for a single-threaded subset of JAVA, called JAVA CARD Dynamic Logic (DL) [Bec01, BS01a]. Regarding induction, this theorem prover conforms to the situation described above; the calculus offers a rule for (first order) Peano induction, and therefore it is an ideal candidate for improvement. Furthermore, it is important to note that the method described in this paper as such is not limited to any particular theorem prover.

Overview of the paper: First we explain why problems arise, both motiva-

---

<sup>1</sup>The paper [HW04] constitutes the first part of this thesis.

tion for using partitioned induction rules and where our first idea [HW04] fails, section 2. Then follows a description of the method and its main ideas: use updates of induction variables as source of partition, create the customised induction rule in *destructor style*, see section 5, and use a theorem prover to derive the partition, section 6. In section 7 we discuss soundness and adjustment of the method to make it sound. We show the method at work with a more complex example in section 8. Finally, we make a comparison to Noetherian induction 9, and state conclusions in section 11 and future work in section 12.

## 2 Motivation

We start by looking at the idea of computing customised induction rules based on partition analysis, presented in [HW04]. In short, the idea is that by using partition analysis the proof can be simplified, by breaking the original proof obligation up into several smaller proof goals, each covering a subset of the program – an example of the *divide and conquer* principle. The partition is assumed to be produced by one of several methods from the field of *partition testing*, for instance [RC85, LPU02]. When doing partition testing, a program’s possibly infinite input space is divided into a finite number of disjoint subdomains and picking one or more elements from each subdomain for systematic testing. This technique from software testing is used in [HW04] to compute a partition of the domain of the induction variable, and then the partition is used to derive (mechanically) an induction rule.

The following example is handled very well by the proposed method; after application of the partitioned induction rule, the resulting proof requires only minimal interaction from the user. However, we show some weaknesses which provide motivation for this paper. The variable  $c$  in the example is assumed to be a compile time constant and thus known to the prover.

**Example 1.** *Motivating example,  $\phi(i)$ :*

$$\begin{aligned}
 & i \geq 0 \wedge c \geq 1 \rightarrow \\
 & \langle \text{while } (i > 0) \{ \\
 & \quad \text{if } (i \geq c) \{ \\
 & \quad \quad i = i - c; \\
 & \quad \} \text{ else } \{ \\
 & \quad \quad i--; \\
 & \quad \} \\
 & \rangle \rangle i = 0
 \end{aligned}$$

Here the proof goal is  $\forall i \in \mathbb{N} \cdot \phi(i)$ , which requires the use of induction to be proved. Rather than using the standard (Peano) induction rule, the proposed method computes a partition induction rule that is tailor-made for the particular problem at hand. The concern of the method is to prove total correctness of loops (in example 1 this means to prove that starting in a state where  $i \geq 0 \wedge c \geq 1$  holds, the loop will terminate in a state where  $i = 0$  holds) and the first step

is to find a partition of the domain of the induction variable ( $i$ ). Using the method for the above example the result is the specialized base cases and the induction step below.

$$\phi(0) \tag{1}$$

$$\phi(1) \wedge \dots \wedge \phi(c-1) \tag{2}$$

$$\forall n \in \mathbb{N} \cdot \phi(n) \rightarrow \phi(n+c) \tag{3}$$

The induction variable is  $i$  and its domain is  $\mathbb{N}$ . Each of the three proof obligations are derived from the partition of the induction variable induced by the *branch predicates*, in the example  $x \leq 0$ ,  $x > 0 \wedge x < c$  and  $x \geq c$ . The method constructs base cases from all finite subdomains and step cases for the infinite ones. For further details on the method or the derivation of this particular example, refer to [HW04]. Combining these proof obligations into an induction rule in the JAVA CARD DL sequent calculus (we will describe the essentials of the logic later), we get

$$\frac{\Gamma \vdash \phi(0) \wedge \dots \wedge \phi(c-1) \quad \Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i) \rightarrow \phi(i+c)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

The semantics of this rule is that if the two premises hold in a state, then the conclusion is true in that state. In particular, if the two premises are valid, then the conclusion is valid. In practice, rules are applied from bottom to top: from the old proof obligation, new proof obligations are derived. This rule makes the induction proof very simple, requiring only minimal interaction from the user, and it could in principle be automated. The most important reason for why the proof becomes simple is because in the step case  $\forall i \in \mathbb{N} \cdot \phi(i) \rightarrow \phi(i+c)$ , when the part of the loop body that decreases the induction variable with “ $i = i - c$ ;” is symbolically executed, it cancels out the corresponding “ $i + c$ ” increase of the induction variable and the proof goal becomes valid trivially. In contrast, using ordinary Peano induction with a step  $\phi(i) \rightarrow \phi(i+1)$  results after symbolic execution in  $\phi(n) \wedge n \geq c \rightarrow \phi(n+1-c)$  – a proof obligation requiring induction in turn.

However, besides the benefit of using the described customised rule, there is an inherent complication in the construction of it – the branch predicates play a crucial role in the approach in [HW04]. In fact it is required that the branch predicates somehow capture what is being computed in the corresponding branch for the resulting induction step to really provide a simplification. This is not always the case and if the branch predicates are completely unrelated to the induction variable, we simply get no information from them on how to partition the domain of the induction variable. For instance in array-sorting algorithms, it is common that the induction goes over the indexes of the array, but the branch predicates typically have a comparison between the elements to be sorted and these might be randomly ordered. Furthermore, the process of transforming general branch predicates into predicates of the form that the method requires is non-trivial, in particular for the process to be mechanized. It involves finding the inverse of functions.

Rather, we would like to create the partition based on more adequate information embedded in the structure of the program than the branch predicates. The particular idea we will work out here is to capture and use the modifications performed by the loop body on the induction variable as the basis for the partition. Finally we would like to avoid the complication of finding inverses to functions. These two improvements are the main concerns of this paper.

### 3 Preliminaries

Here we describe the essentials of the logic and theorem prover that we have been using for this work. However, it is important to note that the method described in this paper as such is not limited to any particular theorem prover.

In this work, the same theorem prover serves both as the subject of improvement as well as the implementation basis for automation of the method. It is a component of a larger verification system for which the target users are software engineers. With the help of the system users can create UML models, programs and constraints, and then formally analyze these. The system, called KeY [ABB<sup>+</sup>04], is integrated into a commercial CASE-tool and has been created with the outspoken aim of being a software-engineer-friendly tool for formal methods. As the programming language the system currently uses a single-threaded subset of JAVA, called JAVA CARD [Che00].

In the stand-alone theorem prover that we will focus on, the verification paradigm is to execute programs with symbolic values, which then are checked (symbolically) against the formal specification. For a background reference to symbolic execution, see [HK76]. The logic used in the prover is Dynamic Logic [HKT00], abbreviated DL. This DL has been extended specifically for the use of JAVA CARD, to a logic called JAVA CARD DL [Bec01, BS01a]. Dynamic logic is a first-order logic with modalities for partial,  $[p]$ , and total correctness,  $\langle p \rangle$ , where  $p$  is a JAVA CARD sequence. (More intricate modalities have been developed as well, see [BM03, BS01b].) For instance, the formula  $\phi \rightarrow \langle p \rangle \psi$  is valid if for every state satisfying the precondition  $\phi$ , a run of the program  $p$  starting in that state terminates (normally) in a state where the postcondition  $\psi$  holds. Dynamic logic allows to include program statements in the formulas for representation of states. Sequents in the calculus are notated using the scheme  $\phi_1, \dots, \phi_m \vdash \psi_1, \dots, \psi_n$  which has the same semantics as the formula  $\phi_1 \wedge \dots \wedge \phi_m \rightarrow \psi_1 \vee \dots \vee \psi_n$ . An example of a deduction rule in the calculus is the rule for conjunction:

$$\frac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}$$

This rule gives rise to two *branches*, (the rules are applied from bottom up) because the rule introduces two new *proof obligations*,  $\phi$  and  $\psi$ , that must be closed in order to conclude that  $\phi \wedge \psi$  holds. Two other rules that we make use

of when proving total correctness for loops, are the following:

$$\frac{\Gamma \vdash \psi \quad \Gamma \vdash \langle \alpha; \text{while}(\psi) \alpha; \rangle \phi}{\Gamma \vdash \langle \text{while}(\psi) \alpha; \rangle \phi} \quad (\text{R1})$$

$$\frac{\Gamma \vdash \neg\psi \quad \Gamma \vdash \phi}{\Gamma \vdash \langle \text{while}(\psi) \alpha; \rangle \phi} \quad (\text{R2})$$

These are the rules we use after the application of an induction rule, to *unwind* the body of the loop.

Also worth noting, there is a mechanism in JAVA CARD Dynamic Logic to simplify the treatment of exceptions, aliasing, and other side effects, called *updates*, that we later make heavy use of in the construction of induction rules. These updates are essentially pairs of program location names and side effect free expressions. The updates appear just before a modality and accumulate changes as the symbolic execution proceeds. An update is of the form  $\{x := t\}$  where  $x$  is a program variable and  $t$  is a term and the intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e.  $\{x := t\}\phi$  has the same semantics as  $\langle \mathbf{x} = \mathbf{t}; \rangle \phi$ . The updates are used to handle aliasing, which is achieved by calculus rules for simplifying updates, or branching the proof as necessary, for a more in-depth description of the updates and JAVA CARD Dynamic Logic see [Bec01, BS01a].

As an example, consider the proof obligation  $\{i := 10\}\langle \mathbf{i} = \mathbf{i} - 5; \rangle i = 5$ . The  $\{i := 10\}$  demonstrates the syntax for an update, and the meaning is that in the current state the value of the program variable  $i$  is equal to 10. Applying the available calculus rules for this example leads to symbolic execution of the subtraction and the assignment and thus creating a new update, and the proof obligation transforms into:  $\{i := 10\}\{i := i - 5\}\langle \rangle i = 5$ . Then simplifying the two consecutive updates to  $\{i := 5\}$  and removing the now empty modality  $\langle \rangle$  leads to the formula:  $\{i := 5\}i = 5$  the value in the update for  $i$  can be substituted into the postcondition since the modality is gone. The reason is that it is only the modalities that change program variables, if there is no modality in the formula the value of the program variable is fixed (or rigid).

In JAVA CARD DL one cannot quantify over a program variable. This restriction is required because a program variable can change its value during symbolic execution, or in other words, it is not *rigid*. Therefore we have to introduce a logical variable ( $il$ ) that can be quantified over and we simply assign its value to the program variable in an initial update, for instance  $\forall il \in \mathbb{N} \cdot il \geq 5 \rightarrow \{i := il\}\langle \mathbf{i} = \mathbf{i} - 5; \rangle i \geq 5$ .

Furthermore, we use some limitations from the start; these are concerning data structures, number systems, programs and modalities. The data structures we consider are natural numbers and integers, and we use  $\mathbb{N}$  and  $\mathbb{Z}$  to denote these sets. With integers we mean the ordinary *mathematical integers* of infinite range. This is to keep the presentation simple since limiting the range makes the proofs more complicated. There are facilities in the prover to deal with real JAVA CARD integers, but we will not use them here. For more details on the

number system, see [BS04]. The programs we consider in this paper are focused to the `while` loop construct, and as a simplification we did not use nested loops. Regarding modalities, only total correctness is considered, partial correctness is mentioned, but no esoteric transaction or trace modalities [BM03, BS01b]. Apart from these restrictions the full range of the `JAVA CARD` is treated.

## 4 Induction

Mathematical induction is an essential tool for reasoning about iterative and recursive structures and formulas. It allows us to once and for all prove properties of some iterative sentence for all instances in a domain. The name induction is sometimes confusing because in all other sciences apart from maths it means to form a conjecture from a sample set of evidence, i.e. not a formal proof of any kind. In mathematics it is however a deductive process that does establish a formal proof of some conditions over a possibly infinite domain.

We have briefly seen the special case of induction over the natural numbers, also known as first order Peano-induction. Here it is in `JAVA CARD DL`:

$$\frac{\Gamma \vdash \phi(0) \quad \Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i) \rightarrow \phi(i+1)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

$\phi(i)$  is an arbitrary logical formula that may contain the variable  $i$  and  $+$  is mapped to the successor operation for natural numbers. A proof of the soundness of this can be found for example in the book [Hei95].

This form of induction allows us to prove a formula valid for all natural numbers, however, how difficult the proof becomes is highly dependent on the particular formula at hand. For instance, the following formula is very well suited for Peano-induction: `{i := il}{while(i > 0) i--;} i = 0`. Using the precondition that the counter starts out as being greater than or equal to zero, this example is simple to prove totally correct since it decrements by steps of one and ends at zero. However, if we alter the program slightly to for instance stepping down by division by 2, it is suddenly very hard to make immediate use of Peano-induction. The user trying to prove the formula valid will have to supply a mapping from the induction variable to the natural numbers, a non-trivial task to say the least, and also obfuscating the resulting proof. For this reason we will look at a more general form of induction.

There exists a generalisation called Noetherian induction, or well-founded induction, that makes it easier to prove the same statement because the induction hypothesis is stronger. Using the well-founded induction principle, proving

$$\forall m \in M \cdot (\forall k \in M \cdot k \prec_M m \rightarrow \phi(k)) \rightarrow \phi(m)$$

and that  $(M, \prec_M)$  is a well-founded set, means that we have verified  $\forall m \in M \cdot \phi(m)$ . This is the most general form of mathematical induction. Using this rule, we would in many cases avoid the need for generalisation of the proof obligation, like in the example decrementing by division by 2. Still, the user-interaction involved in the remaining proof after application of the Noetherian

induction rule, can be very complicated because of the inherent mix of data and control-flow correctness and the size of the proof.

The customised induction rules created in this paper are in fact instances of well-founded induction, as we show later. The concern of our work is in user interaction, rather than in proof strength, and we make a comparison between customised induction rules and Noetherian induction in section 9.

To summarise, our problem is the following: given a program location  $i$ , a logical variable  $il$  (we must have a logical variable to quantify over, since quantification over program variables is not allowed in `JAVA CARD DL`) and a proof obligation on the form  $\forall il : int. \phi(il)$  where  $\phi(il)$  is  $\{i := il\} \langle \text{while}(\xi) \rho \rangle \alpha$  and  $\varphi$  is a precondition, that may be empty (or trivially valid),  $\xi$  is an arbitrary terminating condition of the loop,  $\alpha$  is an arbitrary post-condition and  $\rho$  is an arbitrary loop-free `JAVA CARD` program statement (including blocks of course). Construct a customised induction rule using  $i$  and  $il$  that simplifies the task of proving the original proof obligation.

## 5 Computing Customised Induction Rules

The problem, as we have seen, is to come up with a way to mechanically construct customised induction rules for loops in imperative (`JAVA CARD`) programs. The induction rules we are interested in should feature customised step and base cases, tailored to a particular loop, based on a partition of the domain of the induction variable. The assumption, supported by the results in [HW04], is that this partitioning makes it easier for the user to perform the induction proof and sometimes the process may be completely automated. Branch predicates unrelated to the ideal induction step confuse the old method and we want to ensure that relevant information is used instead.

In the examples in this section we omit the updates and logical variables that would be necessary in `JAVA CARD DL`. Furthermore we only consider the domain of the natural numbers, when nothing else is stated, and assume that the operators are defined properly for this domain. (In particular we assume that the `'-'` operator is defined in terms of the predecessor function  $p(n)$  and that  $0 = p(0) = p(\dots p(0))$ . As this is not the case in `JAVA CARD DL` we will address this in later sections, and extend the results to the full integer range.) In order to start the search for a customised induction rule we consider again a very simple example.

**Example 2.** *Decrementing to zero*

$$\begin{aligned} \phi(i) &\leftrightarrow \\ i \geq 0 &\rightarrow \\ &\langle \text{while } (i > 0) \{ \\ &\quad i--; \\ &\} \rangle i = 0 \end{aligned}$$

This program decrements the variable  $i$  to zero. It is easily verified with the standard induction rule. Recall the step rule for standard induction:

$$\forall i \in \mathbb{N} \cdot \phi(i) \rightarrow \phi(i + 1)$$

where  $\phi(i)$  is the proof obligation from example 2. The standard way to proceed with the proof is to apply the while rule to unwind one iteration of the loop on the right hand side of the implication. The body of the loop is then symbolically executed and the “ $i--$ ” statement cancels the “ $+1$ ” increase of the induction variable in the step, left and right hand sides of the implication become syntactically equivalent, leading to closure of the proof-branch.

From this we make the informal and tentative conjecture that the specialized induction rules we construct should have this property that the step(s) is canceled out by the loop body. Furthermore, example 2 terminates when  $i$  reaches zero, thus matching the base case perfectly. This leads us to a second guess, namely that the base case should be the domain of the induction variable dictated by the termination condition of the loop. In summary, to construct this type of induction rule we need: 1) An induction step, which cancels out the work done to the induction variable by the loop, and 2) A base case, which covers what is not in the loop. The final product should be a deduction rule along the lines of:

$$\frac{\Gamma \vdash \forall i \in \mathbb{D}_b \cdot \phi(i) \quad \Gamma \vdash \forall i \in \mathbb{D}_s \cdot \phi(i) \rightarrow \phi(s(i))}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

where

$\mathbb{D}_b$  is the domain of the base case,

$\mathbb{D}_s$  is the domain of the step case,

$s(i)$  represents the (customised) successor function of the step case.

In the following we continue to informally examine some small examples to identify the components that are required for building a customised induction rule. First we consider loops without branching in their bodies and then extend the results to cover branching loops. This process also gathers useful insights into how we can make use of a semi-automatic theorem prover for the construction job.

Now, deriving the successor function for the step case, how do we do that? As previously stated we would ideally want to make use of the update to the induction variable. From example 2 and its successful match with the standard induction rule, we observe that the successor function in the step is the inverse of the function describing the assignment statement in the loop.

We proceed with another example where the standard induction rule is less useful, and try to come up with a customised induction rule for it (Recall our temporary assumption about the ‘ $-$ ’ operator being defined in terms of a predecessor function  $p(n)$  so that  $0 = p(0) = p(\dots p(0))$ ):

**Example 3.** *Decrementing by 2:*

$$\begin{aligned} & \phi(i) \leftrightarrow \\ & i \geq 0 \rightarrow \\ & \quad \langle \text{while } (i > 0) \{ \\ & \quad \quad i = i - 2; \\ & \quad \} \rangle i = 0 \end{aligned}$$

Similarly as before we can see that the assignment to the induction variable here is  $i - 2$  and inverting this gives us the successor function  $s(i) = i + 2$ , and  $\phi(i) \rightarrow \phi(i + 2)$  as a suggested induction step.

To complete the step case in the rule, we also need a domain for the step case. Informally we again study the above example and see that the loop guard  $i > 0$  can serve as the characteristic predicate of the domain,  $\mathbb{D}_s = \{i \in \mathbb{N} \mid i > 0\}$ .

Now that we have a step case and its domain, all we need to be able to finish the customised induction rule is a suitable base case. For this particular example it is trivial, the only number left over for a base case (of the natural numbers) is 0. Thus we conclude that the domain of the base case is  $\mathbb{D}_b = \{0\}$ . So now our customised induction rule looks like this:

$$\frac{\Gamma \vdash \phi(0) \quad \Gamma \vdash \forall i \in \{i \in \mathbb{N} \mid i > 0\} \cdot \phi(i) \rightarrow \phi(i + 2)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

Note that this rule is not sound as the base case does not prove  $\phi$  for all the numbers that are not implied by the step case. However we will not worry about this now as the changes introduced in the next section will fix the problem. However, we should look a little bit more at this process since we have no reason to assume 0 is always going to be the base case.

**Example 4.** *Decrementing to 10:*

$$\begin{aligned} & \phi(i) \leftrightarrow \\ & i \geq 0 \rightarrow \\ & \quad \langle \text{while } (i > 10) \{ \\ & \quad \quad i = i - 2; \\ & \quad \} \rangle i \leq 10 \end{aligned}$$

In this example we can see that the domain of the step case has to be adjusted to  $\mathbb{D}_s = \{i \in \mathbb{N} \mid i > 10\}$ . (Note also that we updated the postcondition for the program still be correct, however this is of no importance in the rule derivation.) Now we will do as hinted before and state that the rest of the natural numbers must belong to the base case; this makes sense because if there existed some number which was outside the step case domain but not inside the base case domain, the induction rule would not be sound. Therefore if we know that the step case domain is defined by some predicate  $P(i)$  such that  $\mathbb{D}_s = \{i \in \mathbb{N} \mid P(i)\}$ , then the base case domain must be defined by the negation of that predicate:  $\mathbb{D}_b = \{i \in \mathbb{N} \mid \neg P(i)\}$ . For example 4 this means that the domain of the base

case is  $\mathbb{D}_b = \{i \in \mathbb{N} \mid \neg(i > 10)\} \leftrightarrow \mathbb{D}_b = \{i \in \mathbb{N} \mid i \leq 10\}$ , and the customised induction rule becomes:

$$\frac{\Gamma \vdash \forall i \in \{i \in \mathbb{N} \mid i \leq 10\} \cdot \phi(i) \quad \Gamma \vdash \forall i \in \{i \in \mathbb{N} \mid i > 10\} \cdot \phi(i) \rightarrow \phi(i+2)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

## 5.1 Destructor Style Induction

So far we have extracted the induction variable update and inverted it to create a successor function for use in the induction step formula. In our examples, quite simple functions occurred. However, this inversion is non-trivial in general since the induction variable might be updated using arbitrary non-linear operators and method calls. For instance, consider the following example, using a quadratic operator:

**Example 5.** *Squared steps:*

```

forall i . i_l > 1 →
  { i := i_l }
  { while (i < 100) {
    i = i * i;
  } } true

```

Here the equation  $i_{n+1} = i_n * i_n$  solving for  $i_n$  yields  $i_n = \sqrt{i_{n+1}}$ , giving rise to the step  $\phi(i) \rightarrow \phi(\sqrt{i})$ . It may not always be clear what this means in a program logic semantic, the square root is usually a built in function defined for floating point numbers. To solve this by introducing a mathematical square root means extra overhead and again it does not map to the operators and definitions for the programming language at hand. In particular for KeY and JAVA CARD DL there is no special handling of square roots, severely complicating such proofs. Also inverting a function automatically is non-trivial, so for this and the above reason we would like to avoid it if at all possible.

The idea is to perform induction as usual, but start “one step earlier”. If we consider the step  $\phi(i) \rightarrow \phi(\sqrt{i})$  and replace  $i$  with  $i^2$  everywhere we end up with the formula  $\phi(i^2) \rightarrow \phi(i)$ , suddenly the need for square roots is gone! The intuitive reason that it will also stay gone is that we are only using the calculations of the program code in the forwards direction, hence they should all be well defined in the logic. This is a big advantage, and we can now handle all sorts of loops that give rise to arbitrary functions, with no requirement of being possible to invert. With the change to destructor style, the deduction rule we are trying to derive takes the form presented below instead.

$$\frac{\Gamma \vdash \forall i \in \mathbb{D}_b \cdot \phi(i) \quad \Gamma \vdash \forall i \in \mathbb{D}_s \cdot \phi(p(i)) \rightarrow \phi(i)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

where

- $\mathbb{D}_b$  is the domain of the base case,
- $\mathbb{D}_s$  is the domain of the step case,

$p(i)$  represents the predecessor (or destructor) function of the step case.

When we derive rules on this form we simply take the same assignment to the induction variable to be the predecessor functions. The technical details of how this is done we will go into later. The rest of the process stays the same.

Returning to example 4 and applying this strategy we arrive at a predecessor function  $p(i) = i - 2$  and the new customised induction rule:

$$\frac{\Gamma \vdash \forall i \in \{i \in \mathbb{N} \mid i \leq 10\} \cdot \phi(i) \quad \Gamma \vdash \forall i \in \{i \in \mathbb{N} \mid i > 10\} \cdot \phi(i - 2) \rightarrow \phi(i)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)}$$

## 5.2 Branching

One of the main driving forces behind this work is trying to simplify the proof of loops that contain branches, or implicit branching cases. It is in the case of a loop that updates the induction variable differently in the different branches of its body that we particularly expect the user interaction to be simplified from a customised induction rule.

The procedure to create rules for programs with branching loop bodies is very similar to what we have shown so far, except that for each assignment to an induction variable we need to find the condition that led to that program statement and use that condition to create the domain of the corresponding step case. We use a first order formula that we call a *branch predicate* to describe exactly those conditions that must be true for a particular program statement to be (symbolically) executed. We use  $BP_n(i)$  to indicate the predicate of the  $n$ th branch. Note that there can be side effects during the symbolic execution along a path, both in evaluating and in between the branching conditions; the effect of those are captured in the branch predicate as well. We illustrate this with an example:

**Example 6.** *Branching loop*

```
while(i > 0) {
  i++;
  if(i % 2 == 1) {
    i = i - 3;
  } else {
    i = i - 2;
  }
}
```

Here the branch predicate for reaching the statement ‘ $i = i - 3$ ;’ is  $BP_1(i) = i > 0 \wedge (i + 1) \bmod 2 = 1$  and for the second branch  $BP_2(i) = i > 0 \wedge (i + 1) \bmod 2 \neq 1$ .

Since every program statement inside a loop has an associated branch predicate, and so have also the particular program statements that update the induction variable, it should come as no surprise that we use them to create our cus-

tomised induction rule. Each of these branch predicates can be used as the characteristic predicate of a subdomain of the induction variable,  $\{i \in \mathbb{N} \mid BP_n(i)\}$ . The base case we define to be anything that is not in one of the step cases, that is for  $n$  step cases:

$$\mathbb{D}_b = \{i \in \mathbb{N} \mid \neg \bigvee_{k=1 \dots n} BP_k(i)\}$$

so our new customised induction rule becomes

$$\frac{\Gamma \vdash \forall i \in \mathbb{D}_b \cdot \phi(i) \quad \Gamma \vdash \forall i \in \mathbb{D}_{s_1} \cdot \phi(p_1(i)) \rightarrow \phi(i) \quad \dots \quad \Gamma \vdash \forall i \in \mathbb{D}_{s_n} \cdot \phi(p_n(i)) \rightarrow \phi(i)}{\Gamma \vdash \forall i \in \mathbb{N} \cdot \phi(i)} \quad (4)$$

where

$\mathbb{D}_b$  is the domain of the base case,

$\mathbb{D}_{s_1} \dots \mathbb{D}_{s_n}$  are the domains for the different step cases (from 1 to  $n$ ), and

each arbitrary  $\mathbb{D}_{s_k} = \{i \in \mathbb{N} \mid BP_k(i)\}$

$p_n(i)$  represents the predecessor function of the  $n$ th step case.

In the above example we had a different assignment statement in each branch, giving rise to two different induction steps. For the first branch the step becomes  $\forall i \in \mathbb{N} \cdot i > 0 \wedge (i+1) \bmod 2 = 1 \wedge \phi(i-2) \rightarrow \phi(i)$ , and for the second  $\forall i \in \mathbb{N} \cdot i > 0 \wedge (i+1) \bmod 2 \neq 1 \wedge \phi(i-1) \rightarrow \phi(i)$  (the formula  $\phi(i)$  is assumed to be a total correctness statement about the loop in the example).

Note that the branch predicates do not necessarily refer to  $i$  at all, it is then a bit unclear what the partition means in terms of the induction variable. There are in this case two alternatives, either the other variables in the predicate are not affected by the loop (they are rigid) or else we actually need to perform induction over several variables. In this paper we will only deal with the former case, but we return to the second possibility in the section on future developments.

So we now have what we need to define our customised induction rule (in loose terms, at least), our method is at present:

1. Find all the assignments to the induction variable.
2. Turn them into predecessor functions on the form  $p_n(i)$ .
3. Find the branch predicates guarding each of these updates,  $BP_n$ .
4. Negate the disjunction of all  $BP$ s to form the base case predicate.
5. Combine these components into an induction rule.

## 6 Mechanizing the Approach Using a Theorem Prover

In this section we describe technically the building of our customised induction rules, employing elements of JAVA CARD DL and the KeY prover to perform

the calculations and derivations. In particular we will describe how to 1) derive the predecessor functions needed for the induction steps, 2) find the partition, or the branch predicates, for the step cases, and 3) construct the base case from the information we have.

The basic idea we use is similar in spirit to what has been termed “the productive use of failure” in [Ire96]. However, the similarity is more one of name than anything else as a complete study of the techniques are outside the scope of this paper. The general idea is that by first running a proof attempt into the ground, we can learn something about the problem, and then try again with a better idea (in our case a better induction rule). The “learning” part does not imply learning in its ordinary computer science sense, the process is more analogous to ripping an object into its constituent parts to figure out how it works and then putting it back together to be able to work better with it. This is in many ways similar to what is used in the thesis [Pla04] to construct strongest specifications, although our needs are simpler.

## 6.1 Finding Predecessor Functions

Our logical framework (JAVA CARD DL) keeps track of assignments to variables (or program locations) in the *updates* that accumulate in front of a modality as symbolic execution goes on. We have earlier established that we need these assignments to create customised induction steps, and we should be able to make use of these updates.

We now revisit example 3 but this time extended to the entire integer domain and with the required quantifiers and initial update. Recall that in JAVA CARD DL one cannot quantify over a program variable and therefore we have to introduce a logical variable (*il*) that can be quantified over and assign its value to the program variable in the initial update (in curly brackets).

**Example 7.** *Decrementing by 2, again:*

$$\begin{aligned} \vdash \forall il \in \mathbb{Z} \cdot il \geq 0 \rightarrow \\ \{i := il\} \\ \langle \text{while } (i > 0) \{ \\ \quad i = i - 2; \\ \} \rangle i = 0 \vee i = -1 \end{aligned}$$

To be able to extract useful information from this loop we must process it in some way. We would like to use the machinery of our theorem prover (KeY) for this job. We have observed that when symbolically executing a loop body, then whatever it does to the induction variable is recorded in an update. So to attain this information we simply apply the calculus rule to unwind the loop once and simplify (automatically). We can be quite sure that the proof attempt *will* get stuck, because if it does not then we did not need induction in the first place. Once we have finished applying rules we will be left with an intact copy of the original loop, together with a set of updates and conditions. For the code in example 7 it would look like this:

**Example 8.** *Decrementing by 2, stuck after unwinding the loop:*

$$\begin{array}{l}
 il_c > 0 \\
 \vdash \\
 \{i := il_c - 2\} \\
 \langle \text{while } (i > 0) \{ \\
 \quad i = i - 2; \\
 \} \rangle i = 0 \vee i = -1
 \end{array}$$

In this case our expectations seem justified as we have arrived in a position where the information about assignments to the induction variable is in a convenient place, namely in the update in front of the loop. (The logical variable  $il$  has been replaced by a constant  $il_c$  during the application of the rule for quantifier elimination.)

If we assume (or require) that there is an update involving the induction variable before we start the loop-unwinding and that this update has only one logical variable, such as  $\{i := il\}$ . Then to construct the destructor function all we have to do after symbolic execution is to find the update that has  $i$  as the left hand side:  $\{i := \alpha\}$ . Since  $\alpha$  is a side-effect free expression that contains the logical induction variable  $il$  (or a constant  $il_c$ ), we can then express it as a function of  $il$  and this is in fact our predecessor function for the corresponding steps (using substitution of  $il$  in  $\alpha$ ):  $p(i) = [il/i]\alpha$ .

To summarize:

1. Find the update that contains the induction variable  $il$ :  $\{i := il\}$ .
2. Unwind the loop once and run a strategy to automatically apply calculus rules until the proof attempt gets stuck.
3. Find the update that contains the induction variable  $\{i := \alpha\}$ .
4. Take the predecessor function to be  $p(i) = [il/i]\alpha$

A pleasant side effect of letting the symbolic execution do the work for us is that it handles very well cases like the program below.

**Example 9.** *Double assignments:*

$$\begin{array}{l}
 \vdash \\
 \{i := il\} \\
 \langle \text{while } (i > 0) \{ \\
 \quad i = i - 3; \\
 \quad i = i + 1; \\
 \} \rangle i = 0 \vee i = -1
 \end{array}$$

The (automatically applicable) rules for symbolic execution and update simplification will reduce it to have exactly the same result as in example 8.

## 6.2 Extracting the Domain

We also need the domain for the step case to be able to construct our induction rule; earlier we have just 'seen' what conditions the update is guarded by. Now we must show how this can be done mechanically.

We recall the stuck proof attempt shown in example 8:

$$\begin{array}{l}
 il_c > 0 \\
 \vdash \\
 \{i := il_c - 2\} \\
 \langle \text{while } (i > 0) \{ \\
 \quad i = i - 2; \\
 \} \rangle i = 0 \vee i = -1
 \end{array}$$

If we ignore the modality with its update, then what we have left is some boolean formula, in this case  $il_c > 0$  on the left hand side. This predicate will, as it happens, generate exactly the domain we are looking for. This should be no surprise; the symbolic execution has to introduce exactly these conditions in order to symbolically execute the statement guarded by the condition.

So in order to construct the domain predicate (we call it  $BP(i)$ ) we want to gather the terms that are not modal operators in the sequent (the open proof branch where we got stuck), giving  $BP(i) = [il_c/i](a_1 \wedge \dots \wedge a_k \wedge \neg b_1 \wedge \dots \wedge \neg b_l)$  where  $a_1 \dots a_k$  are formulas in the antecedent and  $b_1 \dots b_l$  are formulas in the consequent of the sequent. Additionally, filtering out the modal operator and its update, helps readability a lot.

## 6.3 Finding Partitions

Now we must return to the case where there are several branches inside the loop body. By using the theorem prover to derive partitions, we will be completely independent of external tools.

The process is very simple, we just repeat what we just did for loops without branches. If there are branches inside the loop body these will result in several open proof branches when we get stuck. We now extract the step cases and domain predicates (now we call them branch predicates) for each open branch in turn, labeling them  $p_n(i)$  and  $BP_n(i)$ .

The base case requires no special consideration but is as before constructed from the base domain with the step case domains removed. We can now express this in terms of the branch predicates. If we assume we have  $n$  step cases with the associated branch predicates  $BP_n(i)$  then the base case domain is generated by the predicate  $BC(i) = \neg(BP_1(i) \vee \dots \vee BP_n(i))$ .

## 7 Soundness

In this section we discuss the soundness of our customised induction rules. The basic idea is to show that our rule is an instance of well-founded induction,

paying some special attention to the minimal elements. We will need to add some extra proof obligations to our rule to ensure that it is sound. Note that for space reasons we have in this section omitted the type of the quantified variables, which is *integer*. Recall our customised induction rule so far:

$$\frac{\Gamma \vdash \forall i \cdot BC(i) \rightarrow \phi(i) \quad \Gamma \vdash \forall i \cdot BP_1(i) \wedge \phi(p_1(i)) \rightarrow \phi(i) \quad \dots \quad \Gamma \vdash \forall i \cdot BP_n(i) \wedge \phi(p_n(i)) \rightarrow \phi(i)}{\Gamma \vdash \forall i \cdot \phi(i)} \quad (5)$$

where  $BC(i) \leftrightarrow \neg BP_1(i) \wedge \dots \wedge \neg BP_n(i)$ . Recall also that using the well-founded induction principle, proving

$$\forall m \in M \cdot (\forall k \in M \cdot k \prec_M m \rightarrow \phi(k)) \rightarrow \phi(m) \quad (6)$$

and that  $(M, \prec_M)$  is a well-founded set, means that we have verified  $\forall m \in M \cdot \phi(m)$ .

For our rule we want to establish the well-foundedness of the step order; this order is the same as the execution order (or sequence of program states). Showing well-foundedness in this case is the same as proving termination – a very difficult problem for which we cannot hope to come up with a general solution here. Therefore we have instead opted to introduce additional proof obligations in the induction rules we create, restricting the predecessor functions and induction variables to a domain for which well-foundedness is relatively simple to establish.

The limit we have chosen imposed on the method only allows treating predecessor functions and base case predicates such that all steps go in the same direction (in the ordinary integer  $<$  sense) and that direction is towards the base case. If we for the time being only allow predecessors that decrease the argument, the additional proof obligation looks like this:

$$(\forall i \cdot BP_1(i) \rightarrow p_1(i) < i) \wedge \dots \wedge (\forall i \cdot BP_n(i) \rightarrow p_n(i) < i) \wedge \quad (7)$$

$$\forall i, j \cdot BC(i) \wedge \neg BC(j) \rightarrow i < j \quad (8)$$

It is also very important that the base case is not empty, and with the method we have developed so far this is all too possible. A trivial example of this problem is the loop: `while(true) --i;`. We enforce this requirement by introducing a proof obligation demanding that it is shown that there exists some number in the domain of the base case:

$$\exists i \cdot BC(i) \quad (9)$$

We will now show that these two proof obligations together establish a proof of the well-foundedness of the set that we want to perform induction over.

**Lemma 1.** *The proof obligations in formulas (7), (8) and (9) are sufficient to ensure that the set  $(\mathbb{Z}, \prec)$  is well founded, where  $i \prec j \leftrightarrow BP_1(j) \wedge p_1(j) = i \vee \dots \vee BP_n(j) \wedge p_n(j) = i$ .*

*Proof.* By contradiction, assume that the formulas (7), (8) and (9) hold but  $(\mathbb{Z}, <)$  is not well-founded. Then there exists an infinitely descending sequence  $\dots < i_2 < i_1 < i_0$ . According to the definition of the relation  $<$  and formula (7), then there must be an infinite sequence  $\dots < i_2 < i_1 < i_0$ . Now formula (9) says that the base case is non-empty and formula (8) requires that all elements in the base case are smaller than any element in a step case. Therefore there must exist some largest element  $n \in \mathbb{Z}$  for which  $BC(n)$  and such that  $\forall i \in \mathbb{Z} \cdot \neg BC(i) \rightarrow i > n$ . We also know that any set  $(\{i \in \mathbb{Z} \mid i \geq m\}, <)$  is well-founded. Then we can select  $m = n + 1$  to define the well-founded set  $(\{i \in \mathbb{Z} \mid \neg BC(i) \wedge i \geq m\}, <)$ . Since the infinitely descending chain  $\dots < i_2 < i_1 < i_0$  would belong to this set, we have reached a contradiction and we conclude that  $(\mathbb{Z}, <)$  is well-founded.  $\square$

After the modifications to the rule in order to achieve soundness, and also introducing the concept of iterating up as well as down, our new customised induction rule becomes:

$$\begin{array}{c}
\Gamma \vdash \forall i \cdot BC(i) \rightarrow \phi(i) \\
\Gamma \vdash \forall i \cdot BP_1(i) \wedge \phi(p_1(i)) \rightarrow \phi(i) \quad \dots \quad \Gamma \vdash \forall i \cdot BP_n(i) \wedge \phi(p_n(i)) \rightarrow \phi(i) \\
\Gamma \vdash (\forall i \cdot \bigwedge_{k=1 \dots n} BP_k(i) \rightarrow p_k(i) < i) \wedge \forall i, j \cdot BC(i) \wedge \neg BC(j) \rightarrow i < j \vee \\
\Gamma \vdash (\forall i \cdot \bigwedge_{k=1 \dots n} BP_k(i) \rightarrow p_k(i) > i) \wedge \forall i, j \cdot BC(i) \wedge \neg BC(j) \rightarrow i > j \\
\Gamma \vdash \exists i \cdot BC(i) \\
\hline
\Gamma \vdash \forall i \cdot \phi(i)
\end{array} \tag{R3}$$

where  $BC(i)$ ,  $BP_k(i)$ , and  $p_k(i)$  are derived as before.

**Theorem 2.** *Our customised induction rule (R3) is sound.*

*Proof.* We will show that (R3) is an instance of well-founded induction (6) and thus sound. Assume that our induction rule has  $n$  step cases. First, for  $<_M$  in (6) we pick the following order relation:

$$i < j \leftrightarrow BP_1(j) \wedge p_1(j) = i \vee \dots \vee BP_n(j) \wedge p_n(j) = i$$

Then we pick the induction set to be  $(\mathbb{Z}, <)$ . According to lemma 1 this set is well-founded if the two last premise sequents of (R3) are proved. For the other premises we can divide  $\mathbb{Z}$  into subsets (base case  $\{i \in \mathbb{Z} \mid \neg BP_1(i) \wedge \dots \wedge \neg BP_n(i)\}$  and step cases  $\{i \in \mathbb{Z} \mid BP_1(i)\} \dots \{i \in \mathbb{Z} \mid BP_n(i)\}$ ) and look at one case at a time.

- *Base case:* Starting with well-founded induction (6) for the base case:  $\forall m \in \mathbb{Z} \cdot \neg BP_1(m) \wedge \dots \wedge \neg BP_n(m) \rightarrow (\forall k \in \mathbb{Z} \cdot k < m \rightarrow \phi(k)) \rightarrow \phi(m)$ . Since no element in  $\mathbb{Z}$  is related by  $<$  to an element for which no BP is true, we get:  $\forall m \in \mathbb{Z} \cdot \neg BP_1(m) \wedge \dots \wedge \neg BP_n(m) \rightarrow \phi(m)$ . This is just the base case premise of the rule.
- *Arbitrary (ith) step case:*  $1 \leq i \leq n$  and we start with (6) again:  $\forall m \in \mathbb{Z} \cdot BP_i(m) \rightarrow (\forall k \in \mathbb{Z} \cdot k < m \rightarrow \phi(k)) \rightarrow \phi(m)$ . After inserting the definition of the relation  $<$  we get:  $\forall m \in \mathbb{Z} \cdot BP_i(m) \rightarrow$

$(\forall k \in \mathbb{Z} \cdot BP_1(m) \wedge p_1(m) = k \vee \dots \vee BP_n(m) \wedge p_n(m) = k \rightarrow \phi(k)) \rightarrow \phi(m)$ .  
 Given that there cannot be undefined terms in JAVA CARD DL and that the BPs induce a disjoint partition:  $\forall m \in \mathbb{Z} \cdot BP_i(m) \wedge \phi(p_i(m)) \rightarrow \phi(m)$ .  
 This is just the  $i$ th step case premise of the customised induction rule (R3).

□

We note that the minimal elements of  $(\mathbb{Z}, <)$  are covered by the base case.

## 8 A More Complex Example

In this section we apply the method to Russian multiplication and see how it performs. A JAVA CARD program that performs Russian multiplication is shown below

**Example 10.** *Russian Multiplication:*

$$\begin{aligned} & \forall b_0 \cdot \forall z_0 \cdot a_0 \geq 0 \rightarrow \\ & \{a := a_0\} \{b := b_0\} \{z := z_0\} \{ \text{while } (a \neq 0) \{ \\ & \quad \text{if } (a \% 2 \neq 0) \{ \\ & \quad \quad z = z + b; \\ & \quad \} \\ & \quad a = a / 2; \\ & \quad b = b * 2; \\ & \} \} z = z_0 + a_0 * b_0 \end{aligned}$$

When applying the method for computing a customised induction rule to this program we need to tell it that the program variable  $a$  and its logical counterpart  $a_0$  is the induction variable. The method then produces an induction rule as expected. The rule contains two different step cases, deriving from the branching statement `if (a % 2 != 0)`. Letting  $\phi(i)$  be the formula in the example with  $i$  substituted for  $a_0$  we get the following induction rule.

$$\begin{array}{c} \Gamma \vdash \forall i \cdot i \leq 0 \rightarrow \phi(i) \\ \Gamma \vdash \forall i \cdot i > 0 \wedge i \bmod 2 \neq 0 \wedge \phi(i/2) \rightarrow \phi(i) \\ \Gamma \vdash \forall i \cdot i > 0 \wedge i \bmod 2 = 0 \wedge \phi(i/2) \rightarrow \phi(i) \\ ((\forall i \cdot (i > 0 \wedge i \bmod 2 \neq 0 \rightarrow i/2 < i) \wedge \\ \Gamma \vdash (i > 0 \wedge i \bmod 2 = 0 \rightarrow i/2 < i)) \wedge \forall i, j \cdot i \leq 0 \wedge \neg j \leq 0 \rightarrow i < j) \vee \\ ((\forall i \cdot (i > 0 \wedge i \bmod 2 \neq 0 \rightarrow i/2 > i) \wedge \\ (i > 0 \wedge i \bmod 2 = 0 \rightarrow i/2 > i)) \wedge \forall i, j \cdot i \leq 0 \wedge \neg j \leq 0 \rightarrow i > j) \\ \Gamma \vdash \exists i \cdot i \leq 0 \\ \hline \Gamma \vdash \forall i \cdot \phi(i) \end{array}$$

Despite the rather daunting size of the formula, it is very simple to prove. The user interaction in the remaining proof is: 1) For the base case and both step

cases, the loop in right hand side of the implication must be unwound once, 2) We must instantiate the base case check with any number in the base case, in this case 0 will do fine, 3) We can now apply the automated strategy which will close the proof. After these steps have been applied two branches remain open; the two step cases. This is because the postconditions are now non-trivial and require a fair bit of arithmetic manipulation to close. We also have to instantiate the two other variables ( $b_0$  and  $z_0$ ) in the induction hypothesis in order to attain syntactic equivalence to the succedent.

The interesting part about this is to notice that the interaction follow the same pattern as in the simple example; we use the program itself as induction hypothesis and about the only work required after is instantiating, unwinding and arithmetics. It should also be pointed out that using the current version of KeY the deduction rules for modulo-arithmetic and other arithmetics is not strong enough. Meaning we had to add (and prove) some special case identities in order to perform the proof as described. This does not represent a short-coming of the method developed here but is a problem in the theorem prover at large. Thus when the arithmetic capabilities are improved, the automation of inductive proofs will benefit too.

## 9 Comparison to Noetherian Induction

Consider again the (first-order) Noetherian induction rule:

$$\frac{\Gamma \vdash \forall m \in M \cdot (\forall k \in M \cdot k \prec_M m \rightarrow \phi(k)) \rightarrow \phi(m)}{\Gamma \vdash \forall m \in M \cdot \phi(m)}$$

which is sound as the set  $(M, \prec_M)$  is well-founded. Usually a proof using this rule proceeds by instantiating the rule with the relevant values. Then unwinding the loop in the induction conclusion once and running the automated rules until stuck. At this point the user has to examine the open branches, and instantiate the hypothesis where relevant.

When we compare the two methods we find the following differences in how the proofs done with them turn out. We are now not trying to analyse their relative proof-strength but rather the usability and interaction requirements. To improve readability we will refer to the method developed in this paper as partitioned induction, PI in the following, and we abbreviate well-founded induction WFI.

- WFI introduces only one branch into the proof, as opposed to at least 4 for PI. This is because PI “knows” more about the problem and can present the branches “up-front”; the same content has to be proven anyhow but this helps modularise the proof. In PI we have also introduced separate proof obligations for well-foundedness; this makes PI simpler because it separates the different concerns of the proof and if the well-foundedness branch fails, the user knows where the failure stems from, something that can be more than just a little hard to tell in a proof using WFI (where everything is proven together).

- For the step cases the procedure is usually simple using PI, we unwind the right hand side once at the start. Using WFI we do not have the limiting branch predicates, thus we have to let the user discover these branches and then manually instantiating the general induction hypothesis. Knowing where and when the hypothesis should be instantiated can be difficult when the proof stops with a number of open goals in the middle of a big proof.
- Another difference concerning the step cases is that the domains are explicit in PI, as we introduce branch predicates to restrict the domain. This narrows down the branching in the proof. However since the PI algorithm does not simplify the branch predicates before they are introduced this does result in a lot of redundant branching, but this is not a problem in principle.
- The base case is separated out explicitly in PI, whereas it is implicit in the induction hypothesis in WFI. This is often to an advantage for the WFI as this is usually very simple to prove. Using PI requires interaction (unwinding the base case explicitly), but “just happens” using WFI.
- The WFI goes beyond the PI in the application domain; this is because the WFI can handle steps that go *against* the  $<$  direction dictated, as long as the user can prove that it must eventually descend. For example if one step increases the argument by 1 and the next decreases by 2. In principle this can be done for the PI as well, if the induction rule construction unwinds any cases that go against the stream further. Of course it is not easy to know the right direction, but we can solve it by allowing user interaction.

From the above points we can see that the two techniques are quite similar in performance. We would however venture to say that the partitioned induction is more accessible to entry level users as the interaction is mostly performed early in the proof and it is easier to “debug” a possible failed proof attempt. One can see customised induction as a way of making well-founded induction more automatic, useful and deterministic, by extracting more information from the program.

## 10 Related Work

There is an overwhelming mass of work on the mechanization of induction proofs that has been going on for decades. A comprehensive description of this is out of the scope of this paper, however we will give some pointers to relevant areas. First of all, there are two major approaches of mechanizing induction proofs – explicit and implicit induction. In *explicit* induction, proofs are built using an induction principle for which (explicit) rules are incorporated in the proof, for early seminal work see [Bur69, Aub79, BM79] and for good background references see [Wal94, Bun01, Sli97]. Using *implicit* induction [Red90, BKR95, KM87]

one adds the conjecture to be proven to an equation system with existing axioms, then checks for inconsistencies and if none are found the conjecture is concluded to be an inductive theorem. Implicit induction allows a higher degree of automation. In this paper we deal with explicit induction only, as we are concerned with computing induction rules for use in an interactive theorem prover. Whereas some of the mature techniques for explicit induction are quite powerful (having proved many difficult theorems [BM88, BvHSI90, BvHHS90]), they are often restricted to proving properties about programs written in tiny functional programming languages. In this paper we merely scratch on the surface of automating induction proving, but we do so with the somewhat neglected concerns of a real imperative programming language and a shift of focus towards user-interaction.

Since goal of this work is to simplify the user-interaction in theorem proving for imperative programs, we here specify some other attempts at this task. For instance, in the B method [Abr96], loop correctness is proved using five different proof obligations, each with a distinct aim clear to the user. Interestingly, we have noticed that we also achieve some simplified user interaction using the customised induction rules, because the proof is split at an early stage and separating the concerns of the different parts of the proof. However, we are relieved of the burden to annotate the program with an invariant and a variant, as in the B method (we might need to generalise our induction hypothesis, though). The reduction in user interaction that is achieved by separating the concerns of control and data correctness parts of proof goals has also been noticed in the hardware community [SKK94, HB95]. Other pieces of work to ease the user interaction in proving loop correctness are the automatic generation of loop invariants [RCK04a, RCK04b], interactive proof critics [LJR99] and rippling [BSvH<sup>+</sup>93]. Furthermore, ACL2 [BKM96, KM97] constitutes a big effort to create an industrial-strength user-guided automated theorem prover, however we find no particular concern for simplifying proofs by induction there.

## 11 Summary and Conclusions

In this paper we have described a method for deriving customised induction rules for proving the correctness of loops in a semi-interactive theorem prover. The new method uses the updates of the induction variable to derive the partition and induction steps, which is an advantage over the first version in [HW04]. Another improvement is that the new method is extended to deal with integers, as opposed to natural numbers only. By using destructor style induction, we avoid the issue of inverting the step functions, which used to be a major limitation. The customised induction rules created by the method are sound. We showed how to use the theorem prover to compute branch predicates, which represents a big step forwards since we cut off the need for external tools. A prototype implementation of the method has been made and allowed comparison to Noetherian induction.

The method developed in this paper is a step on a path towards automated

proving of loops in KeY. It has achieved a shift of focus for the user interacting with the prover. The method is not stronger than the well-founded induction, but the required interaction steps are moved to an earlier point in the proof and branching provides labels that make interaction more focused (if nothing else this makes writing tutorials easier). There are more branches but they are smaller and the user interaction required within each branch in the rest of the proof is simpler, for instance it is easier to find the correct instantiations. Another important consequence of this is that it is much easier for the user to analyze a failed proof attempt and see what kind of generalization of the postcondition that might be necessary. Since the control flow has been lifted into separate proof branches we can now attack this problem with specialized methods. Far from being a conclusive treatise on the problem we have merely scratched the surface of a whole host of interesting problems. Therefore in the next, and closing, section we will outline some ideas for future developments.

## 12 Future Work

Here we list a number of ideas for future work.

- *Partial Correctness/Box Modality*: In this paper we have focused on proving the total correctness of loops. There is also the option to prove only the partial correctness of a loop (or other program) using the box modal operator,  $\Box$ . For partial correctness, termination is not required. Intuitively, it seems reasonable to use our customised induction rules without the well-foundedness requirement, but that would require some more analysis. However, since proving partial correctness of loops in KeY today requires rather complicated user interaction, for instance the creation of an invariant, there is room for improvement with customised induction rules.
- *Hybrid with Noetherian Induction*: A possibility that arises from the comparison to well-founded induction is to combine the two techniques. By unwinding the body of the loop several times during generation of the induction rule, the resulting rule would be stronger and still keep the modularity of the user interaction.
- *Separate Termination Analysis*: One interesting and possible future development derives in the fact that well-foundedness (of the execution order of the program) and termination of a program are one and the same. If we had an (undecidable) oracle that could tell us if a loop terminates then we could remove the restrictive tests on the induction variable(s). If we had an oracle that could tell us if a loop terminates then we could remove the restrictive tests on the induction variable(s). It would be interesting to exploit practical techniques specialized in this field, for instance in static analysis tools [Pow04].

- *Nested Loops and Multiple Induction Variables:* In proving total correctness of nested loops or induction with multiple variables, we expect no particular differences in user interaction compared to other instances of Noetherian induction. However, it is a limitation of our current work that we did not consider these, and this still has to be done.
- *Other Data Structures:* At present the technique is limited to integers. While this is less of a limitation than the natural numbers, there are other interesting data structures like trees and linked-lists to perform induction over as well. In JAVA CARD and other imperative languages it is legal to create cyclic structures and we are thus required to be more careful when constructing proofs, it is obviously harder to guarantee that the structures are well-founded and thus that induction is valid.
- *Expression Simplification:* For a decent implementation of the algorithm for use by software engineers we would certainly need to simplify the expressions in the branch predicates and especially the base case predicate. This is merely a usability requirement.
- *Towards full Automation:* The algorithm described in this paper represents a step on the way towards automatic proof of loops. There are always obstacles to this, for instance we must devise a way to find the induction variable(s). Furthermore, the interactive steps after the application of the customised induction rule must be eliminated.
- *Generalisation of post-conditions:* Our method removes for many cases the need for generalisation of the induction hypothesis. But there will still be examples where a generalisation of the post condition is required, and even so using the Noetherian induction rule. This problem is simply out of scope in this paper but we see it as a large part of our future work.

## Acknowledgments

Many thanks to Wolfgang Ahrendt and Reiner Hähnle for reading drafts of this paper and for very helpful comments on this work.

## References

- [ABB<sup>+</sup>04] 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 System Modeling*, 2004. Online First issue, to appear in print.
- [Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

- [Aub79] Raymond Aubin. Mechanizing structural induction. *Theor. Comput. Sci.*, 9:329–362, 1979.
- [Bec01] 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.
- [BKM96] Bishop Brock, Matt Kaufmann, and J. Strother Moore. Acl2 theorems about commercial microprocessors. In *FMCAD '96: Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 275–293. Springer-Verlag, 1996.
- [BKR95] Adel Bouhoula, Emmanuel Kounalis, and Michael Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5):631–668, 1995.
- [BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, New York, NY, 1979.
- [BM88] Robert Boyer and J. Strother Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press, 1988.
- [BM03] Bernhard Beckert and Wojciech Mostowski. A program logic for handling Java Card’s transaction mechanism. In *Proceedings, Fundamental Approaches to Software Engineering (FASE), Warsaw, Poland*, LNCS 2621, pages 246–260. Springer, 2003.
- [BS01a] Bernhard Beckert and Bettina Sasse. Handling JAVA’s abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. Technical Report DII 07/01, Dipartimento di Ingegneria dell’Informazione, Università degli Studi di Siena, 2001.
- [BS01b] Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.
- [BS04] Bernhard Beckert and Steffen Schlager. Software verification with integrated data type refinement for integer arithmetic. In *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, LNCS. Springer, 2004. To appear.

- [BSvH<sup>+</sup>93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.
- [Bun01] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.
- [Bur69] R.M. Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12:41–48, 1969.
- [BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The oysterclam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer, Berlin, Heidelberg, 1990.
- [BvHSI90] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer, Berlin, Heidelberg, 1990.
- [Che00] Zhiqun Chen. *JAVA CARD Technology for Smart Cards: Architecture and Programmer's Guide*. JAVA Series. Addison-Wesley, June 2000.
- [HB95] Ramin Hojati and Robert K. Brayton. Automatic datapath abstraction in hardware systems. In *Proceedings of the 7th International Conference on Computer Aided Verification*, pages 98–113. Springer-Verlag, 1995.
- [Hei95] James L. Hein. *Discrete structures, logic, and computability*. Jones and Bartlett Publishers, Inc., 1995.
- [HK76] Sidney L. Hantler and James C. King. An introduction to proving the correctness of programs. *ACM Computing Surveys (CSUR)*, 8(3):331–353, 1976.
- [HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. Foundations of Computing. MIT Press, October 2000.
- [HW04] 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.
- [IJR99] Andrew Ireland, Michael Jackson, and Gordon Reid. Interactive proof critics. *Formal Aspects of Computing*, 11(3):302–325, 1999.

- [Ire96] Andrew Ireland. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.
- [KM87] Deepak Kapur and David R. Musser. Proof by consistency. *Artif. Intell.*, 31(2):125–157, 1987.
- [KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.
- [LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 21–40. Springer-Verlag, 2002.
- [Pla04] André Platzer. Using a program verification calculus for constructing specifications from implementations. Technical report, University of Karlsruhe, Department of Computer Science. Institute for Logic, Complexity and Deduction Systems, 2004. <http://www.functologic.com/logic/Minoranthe.ps>.
- [Pow04] Daniel Powell. Automatic derivation of loop termination conditions to support verification. In *Proceedings of the 27th conference on Australasian computer science*, pages 89–97. Australian Computer Society, Inc., 2004.
- [RC85] D.J. Richardson and L.A Clarke. Partition Analysis: A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, 1985.
- [RCK04a] 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.
- [RCK04b] 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.
- [Red90] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 162–178. Springer, Berlin, Heidelberg, 1990.
- [SKK94] K. Schneider, T. Kropf, and R. Kumar. Control-path oriented verification of sequential generic circuits with control and data path. In *European Design and Test Conference (EDTC)*, pages 648–652, Paris, France, March 1994. IEEE Computer Society.

- [Sli97] Konrad Slind. Derivation and use of induction schemes in higher-order logic. In Elsa L Gunter and Amy Felty, editors, *Proc. 10th International Theorem Proving in Higher Order Logics Conference*, volume 1275 of *LNCS*, pages 275–290. Springer-Verlag, 1997.
- [Wal94] 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.

## Appendix: Pseudo Implementation

Now that we have described both the details of the method as well as the process of discovery, we shall try to formulate the whole process in a more concise way. To do this we formulate the process in pseudo code similar to what we used in our prototype implementation. We explain the auxiliary functions after the pseudo listing.

```

CreateInductionRule( $\lambda$ ,  $i$ ,  $il$ )
1   where
2      $\lambda \leftrightarrow \varphi \rightarrow \{i := il\} \langle \text{while}(\xi) \rho \rangle \alpha$ 
3      $\phi(n) \leftrightarrow [il/n] \lambda$ 
4      $p = \text{CreateProof}(\lambda)$ 
5      $p = \text{ApplyUnwindWhileRule}(p)$ 
6      $p = \text{ApplyStrategy}(p)$ 
7     for each open Goal  $g_k$  in  $p$  do
8       if  $\exists \{i := \delta\} \langle \text{while}(\xi) \rho \rangle \in g_k \cdot il \in \delta$  then
9          $p_k(n) = [il/n] \delta$ 
10         $BP_k(n) = [il/n] \text{GatherPredicates}(g_k)$ 
11      else
12        fail
13     $BC(n) = \neg(BP_1(n) \vee \dots \vee BP_k(n))$ 
14    return CustomisedInductionRule

```

```

GatherPredicates( $g$ )
  where
     $g = a_1, a_2, \dots, a_k \vdash b_1, b_2, \dots, b_l$ 
     $r = \text{true}$ 
  for  $i = 1$  to  $k$  do
    if IsRigid( $a_i$ ) then
       $r = r \wedge a_i$ 
  for  $i = 1$  to  $l$  do
    if IsRigid( $b_i$ ) then
       $r = r \wedge \neg b_i$ 

```

**return**  $r$

Line-by-line explanation of the code:

1. **where**

The where clause is used to define bindings and do pattern matching, in this way it defines what programs the algorithm can handle.

2.  $\lambda \leftrightarrow \varphi \rightarrow \{i := il\} \langle \text{while}(\xi)\rho \rangle \alpha$

Here we constrain the input formula  $\lambda$ ; it contain a total correctness assertion with a while loop on the top level. The formulae  $\varphi, \xi, \rho$  and  $\alpha$  are arbitrary (but of course semantically and syntactically correct).

3.  $\phi(n) \leftrightarrow [il/n]\lambda$

We will need to refer to the input formula parameterized with  $il$ , to this end we define  $\phi$  where  $il$  is replaced with the bound variable  $n$ .

4.  $\mathbf{p} = \text{CreateProof}(\lambda)$

This functions constructs a new proof in the theorem prover, the parameter is the goal formula, that is the formula we wish to establish the validity for. We store the new proof in  $\mathbf{p}$ .

5.  $\mathbf{p} = \text{ApplyUnwindWhileRule}(\mathbf{p})$

Applies the calculus rules for unwinding a while loop (R1) and (R2), see section 3, to the parameter proof. The result is a new proof which we store in  $\mathbf{p}$ .

6.  $\mathbf{p} = \text{ApplyStrategy}(\mathbf{p})$

Executes an automated strategy on the parameter proof, the strategy is a concept in KeY that applies rules until there are no more useful rules to apply. There are several strategies and we assume the one currently called 'Simple Javacard DL without loop unwinding' is used. Again  $\mathbf{p}$  is updated with the result.

7. **for each open Goal**  $g_k$  **in**  $\mathbf{p}$  **do**

We now iterate through each open goal in the proof, the goals which have been closed are ignored as they contribute no new information.

8. **if**  $\exists \{i := \delta\} \langle \text{while}(\xi)\rho \rangle \in g_k \cdot il \in \delta$  **then**

At this step we check if there exists a sub-formula that is an update followed by a while modality, such that  $il$  is contained in  $\delta$ . Note that the symbols used are the ones bound on line 2, meaning that this is a copy of the original loop.

9.  $p_k(n) = [il/n]\delta$

We then take the update to be the  $k$ 'th step function, with  $il$  replaced with a bound variable  $n$ .

10.  $BP_k(n) = [il/n]\text{GatherPredicates}(g_k)$   
Then also the  $k$ 'th branch predicate is defined by the predicates in the open goal, which are gathered in by the function `GatherPredicates`.
11. **else**
12. **fail**  
The algorithm fails to produce a custom induction rule if the conditions on line 8 are violated. If we view the algorithm as a nondeterministic one, we could assume there is a choice point inside `ApplyStrategy(p)` on line 6. This could include going back and letting the user interact with the proof attempt in order to close offending branches.
13.  $BC(n) = \neg(BP_1(n) \vee \dots \vee BP_k(n))$   
We form the base case predicate  $BC(n)$  as described in section 5.2. Here we assume that there were  $k$  open goals.
14. **return CustomisedInductionRule**  
Returns the finished custom induction rule, (R3), the rule is described in detail in section 7.

Finally we also must note the fact that the failed proof attempt from which we extract the information is implicitly discarded at the end of the function. We have no further use for it, the only reason it was introduced was to extract information. The newly created custom induction rule should now instead be introduced into the original proof, where it can be applied to some sub formula (presumably  $\lambda$ ).