Automatic Non-termination Analysis of Imperative Programs

Helga Velroyen

M.Sc. Thesis

Chalmers University of Technology,
Göteborg, Sweden.

Supervisor: Philipp Rümmer
Examiner: Reiner Hähnle

November 2007

Abstract

Software which runs into an infinite loop and thus does not terminate can become a serious problem in real-life software-systems. In this work we developed an algorithm to detect infinite loops in imperative programs. This algorithm uses invariants to prove the non-termination of the target program.

We implemented a software which uses a theorem prover for Dynamic Logic to generate and refine invariants incrementally. The software examines programs of a simple imperative language and works fully automatic. The software was tested on a set of example programs and these tests were successful. We applied the algorithm also on example programs of a object-oriented language and obtained promising results here, too.

To our knowledge, our work is the first algorithm and implementation of a method to prove non-termination of imperative programs.

For M. Lintermanns.

Acknowledgements

I like to thank Prof. Dr. Reiner Hähnle for giving me the opportunity to write my thesis in his research group, the KeY group. My supervisor Philipp Rümmer is a member of this group and I thank him for his constructive and qualified feedback, the proofreading of my drafts and his support during the implementation. I like to thank my colleagues of the KeY group, in particular Mattias Ulbrich for contributing the Takeshi example, Benjamin Weiss for the collaboration concerning external invariant generators, Richard Bubel for his occasional help with KeY and Marcus Baum for his entertaining company in our office.

I thank Prof. Dr. Jürgen Giesl for reviewing my thesis for the RWTH Aachen University and his colleagues of the Research Group Computer Science II, who gave me helpful feedback at my talks.

I like to thank all people who gave me helpful hints to sources for examples and in particular Juri Ganitkevitch who provided the ChaosBuffer example.

There are several people who gave me constructive feedback concerning this document. In particular, I like to thank Tobias Weyand, Petra Welter, Philipp Vorst and Matthias Höller for proofreading parts of my thesis concerning content and style. In particular, I thank Bernd Hauchwitz for reading my complete thesis concerning English grammar and style. I thank Sebastian Stigler for sharing his LaTeX skills.

I thank Jens Forster for his support in resolving the administrative issues which I faced when I planned to write my thesis abroad.

I love to thank my partner Sumedha Widyadharma for simply being there whenever I needed him. In addition, I thank him for his occasional help concerning technical problems.

Ich danke meinen Eltern Brigitte und Konrad Velroyen dafür, dass sie mich in meinem Studium unterstützen und insbesondere dafür, dass sie es mir ermöglichten, meine Diplomarbeit in Schweden zu schreiben.

Ett stort tack till Pernilla Sjöqvist och Mattias Svensson som var min värdfamilj under min tid i Sverige och som lät mig vara en del av deras hem och familj.

Contents

Chapter 1

Introduction

In 1936, Alan Turing proved that no computer program can ever decide for all programs whether they terminate or not. He phrased this problem for turing machines, a simple form of programs which form the base of today's imperative programming languages. The problem is called *the halting problem* and has ever since fascinated computer scientists and mathematicians. Although Mr. Turing has taken all hope of ever solving the problem in general, scientists have started to develop methods to solve at least parts of the problem or the problem restricted to particular classes of programs.

Our research group accepted this challenge and in this work we addressed the problem from the side of non-termination. In our work, we developed an algorithm whose input is an imperative program and whose output is positive if the program does *not* terminate for some of its possible inputs. The description of those critical inputs is also provided by the algorithm. Of course, the algorithm is not complete, which means that not all non-terminating programs can be identified by it.

Non-termination of programs is a bug. Bugs in software can have severe consequences, expressed in the loss of millions of dollars or even human lives. The most common method to search for bugs is testing. Testing means that we start the program in question with specific input values and then observe whether the program works correctly.

A program that works incorrectly terminates with wrong results, terminates abruptly with an exception or does not terminate at all. In general, the disadvantage of testing in comparison to proving is that a positive test result only proves the correctness of the program for the specific input values with which the program was started. In contrast, proving the correctness of a program means showing that the program works correctly for a whole set of input values (if not for all of possible inputs).

A particular drawback of testing concerning non-termination is that it is actually not possible to test a program for non-termination. If we start a program with specific input values and it does not terminate, then we can never determine if the program really does not terminate or just performs a very long (but eventually terminating) calculation. The only property we could test for in this context is whether the program terminates before a given time is expired. All programs which fail this test either terminate after a longer period of time than the given limit or do not terminate at all. In contrast, if we can prove the non-termination of a program for a set of input values, then there is no chance that the program does terminate for any of those inputs.

Improving the quality of software by using formal methods has always been a major goal of the software verification community. There are research projects which examine the termination behavior of programs. We distinguish here between methods to prove the termination and methods to prove the non-termination.

Algorithms for termination proofs exclusively try to find a witness to prove that the program does terminate. If they fail in proving the termination, they do not make a statement about the program's termination behavior. In particular, the algorithms in those situations do not determine if a program does not terminate; they only say that they cannot prove the termination.

In contrast, algorithms for non-termination analyze a program exclusively for non-termination. That means they try to prove the non-termination of a program, but if they fail, they do not state that the program terminates. They just say that they could not prove that the program does not terminate.

Surprisingly, all research projects in the field of termination analysis focus on the proof of termination rather than on the proof of non-termination. Proving termination seems to be more attractive, because termination is the desired behavior of programs. It is only a recent development in the verification community to use formal methods to disprove desired behavior of programs in order to find bugs. Our approach is an analysis method which focusses exclusively on the non-termination of programs.

Termination analysis of term rewriting systems has a very active research community. Term rewriting systems are a simple form of programs, represented by a set of rules to transfer terms into other terms. There is an annual termination competition [MZ07] for software tools which prove or disprove the termination of term rewriting systems. The regular winners are the colleagues of the Research Group Computer Science II, Programming Languages and Verification, of Prof. Giesl at the RWTH Aachen with their tool AProVe [GSKT06].

To our knowledge, the AProVe project is the only project which is also capable of proving the non-termination of programs which can be written as term rewriting systems (see [GSKT05]). The idea behind this approach is to find a term which can be rewritten to a more complex term which contains the original term as subterm. This way, we could apply the rewriting step again and the term grows further and further and thus the program never terminates. In principle, the repeated occurrence of the term as subterm is an invariant for the rewriting step.

There are approaches to translate functional [Swi05], [PSS97] and logical programs [Käu05], [AZ95] into term rewriting systems. The termination proof of the programs is then performed by proving the termination of the respective term rewriting systems. Those translations have the drawback of producing term rewriting systems which are rather big and therefore their termination is hard to prove.

The field of termination analysis becomes more sparse concerning imperative programming, which is the paradigm of the programming languages that are widely used in industrial software development. Sondermann followed the approach of translation into term rewriting systems and presents in [Son06] his translation of a fragment of the Java language. If we can prove the termination of a term rewriting system which is the result of the transformation of a Java program, then we have proven the termination of the original program, too.

The transformation is sound, but unfortunately not complete. That means that we cannot use it to prove the non-termination of a Java program. The problem is that the transformation in some cases transforms a terminating program into a non-terminating term rewriting system. Thus, even if a non-termination checker is able to prove the non-termination, we could not transfer this result to conclude the non-termination of the Java program.

An approach different from the translation into simpler systems is the synthesis of ranking terms or ranking functions [CS01]. Colón and Sipma present in [CS02] a termination analysis approach using those functions. Cook et al. at Microsoft Research developed the tool Terminator ([CPR]), which proves the termination of C programs by generation of ranking functions with the help of abstract interpretation, symbolic execution and separation logic.

None of these approaches for imperative programs actually examines programs for *non*-termination. In cases where the termination cannot be proven, these algorithms output at most the path in the program flow where the problem *might* be. Our algorithm in contrast, proves the non-termination and gives a description of the set of input values which cause the program to not terminate. This information is important in the search for bugs in programs.

We implemented the algorithm for a fragment of Java as the target language, because Java is an example of an imperative, object-oriented programming language and widely used in the industry. Note that the method which we developed is not dependent on any Java-specific features. In principle it could be applied on any other imperative and object-oriented language. Thus, to our knowledge, we developed the first algorithm which proves automatically the non-termination of imperative and object-oriented programs.

Because there is no other software tool like that, there is no standardized example program set either. We built up a database of non-terminating example programs to have a starting point for the comparison of different approaches. In our work, we used it to measure the quality of our implementation and compare different heuristics to each other. We ran a number of experiments on the database and despite the inherent incompleteness of the problem, the results are promising for practical applications.

Our work was done in the scope of the KeY Project, which is a research project at Chalmers Technical University in Gothenburg (Sweden) and other European universities. In this project the KeY prover is developed whose purpose is to verify if programs meet their specifications. Specifications include, among other properties, statements about the termination behavior of a program. Our software is an extension of the KeY prover to prove non-termination in particular.

Organization of this Thesis The following Chapter 2 is a slightly informal introduction into the topic of termination and non-termination of programs, in particular imperative programs. Chapter 3 provides the theoretical background for the understanding of this work. The foundations are followed by Chapter 4, which takes a closer look at how to handle non-termination in dynamic logic.

In Chapters 5 and 6, we present an algorithm of a method to prove non-termination of imperative programs and its implementation as a Java software. Chapter 7 describes a series of experiments on a database of simple While programs and evaluates their outcome. We leave the field of While programs in Chapter 8 and analyze to what degree the algorithm of Chapter 5 can be transferred to programs of a richer programming language, named Heap programs.

We conclude our work in Chapter 9 by summarizing the results, drawing final conclusions, giving pointers to related work and presenting ideas for future work.

Chapter 2

Non-termination of Programs

In this chapter, we introduce the reader to the field of non-termination of programs. We will talk about what exactly non-termination is, how it is caused, why it is a challenge and how we addressed this problem.

2.1 Imperative Programs and their States

Characteristic features of imperative programs are the existence of variables and statements. Variables are used to store values and statements describe the computation of the program. The content of the variables describes the current state of the program. The set of all possible states of a program is called state space.

The statements of the program manipulate the content of the variables and so change the state of the program. This means that programs[1] define transitions from states to states. Thus, a program's state space can be considered as graph with states being the vertices and programs forming (several) transitions.

The execution of a program begins in a start state, in which the variables can have an initial value or might not been assigned one yet (and thus are considered as *undefined*). The execution of a program is a trace through the graph from the start state. If the end of the program is reached, this trace ends in a result state.

2.2 Non-termination of Programs

Non-termination of a program means that its computation never stops. In a graph representing the state space, a non-terminating program is either represented as an infinite path or as a cyclic path. In the case of an infinite path, with each iteration a new state is reached. In case of a cyclic path a finite set of states is visited over and over again in the program's execution.

[1] Because program statements are one-line programs, they are included here, too.

```
path(int i) {
  int a = 0;
  while (i < 10) {
    a += i;
  }
}
```

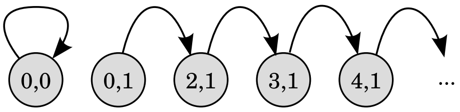


Figure 2.1: Path

This program does not terminate for all input values less than 10. The graph shows a part of the state space of the program Path. The first value in a state is the value of `a` and the second one of `i`. When the program is started with `i` $= 0$, then the program does not terminate. The path of the program then forms a cycle around the state $(0, 0)$. If the program is started with `i` $= 1$ as input value, the program does not terminate either but this time the path is an infinite path starting with the state $(0, 1)$.

2.3 Causes of Non-termination

There are different causes for the non-termination of a program. The most obvious one are infinite loops. Loops are infinite, if their exit criterion, which is the falsification of the loop condition, is never fulfilled.

Other causes of non-termination in programs are infinite recursions or deadlocks. The examination of those is beyond the scope of our work; we focussed exclusively on infinite loops. Actually, recursive programs can always be rewritten into non-recursive programs which use data structures like stacks. The Heap programming language, which we introduce in Chapter 8, is rich enough to use stacks and thus includes recursive programs implicitly. Anyway, structures like stacks could also be encoded as integers, which means that even While programs could contain this functionality.

2.4 The Objective of Non-termination Detection

Non-termination is usually an undesired behavior of programs. There are hardly any applications which are designed to be non-terminating (at least for maintenance there is always a way to shut a system down). This means non-termination is a bug, and bugs need to be found in order to produce good software.

An example for an erroneous program which contains an infinite loop is the program Gauss in Figure 2.2. Here the programmer overlooked the possibility that the input variable `n` can be negative. In this case, the `while` loop does not terminate.

Our work's objective is to provide a method to detect non-termination of programs with respect to the long-term goal of improving software development. That means we would like to find programming errors which lead to non-termination rather than solving the halting problem in general. In particular, we like to achieve

```
gauss(int n) {
  int sum = 0;
  while (n != 0) {
    sum += n;
    n--;
  }
}
```

Figure 2.2: Gauss

This is an example of a While program. The program calculates the Gaussian sum, which is the sum of all integers between 1 and the value of the parameter `n`. It does not terminate for negative input values.

the following goals.

1. Identify non-terminating programs.
2. Identify the critical inputs. Those are the ones for which a program does not terminate.
3. Describe the set of critical inputs as general as possible.
4. Automate 1.-3. as much as possible.

Alan Turing presented in [Tur36] the famous *halting problem*. The halting problem asks the question: Given the description of a target Turing machine and an input sequence, is there a Turing machine that could determine if the target Turing machine will terminate when processing the input? Alan Turing proved in the same work, that there is no such Turing machine.

Modern imperative programming languages are based on the concept of Turing machines. Thus, our goals are dependent to the halting problem and thus they are as unsolvable in general as the halting problem itself. A consequence of the halting problem is that we for example will never be able to identify all non-terminating programs nor all critical inputs of all programs. Nevertheless, although the task is not solvable in general, it is a reasonable objective to solve it at least for as many programs as possible and in particular for programs, which are realistically developed in real-life software projects.

In conclusion, our focus is to help developers to find bugs automatically and thereby improve the quality of software rather than on the (pointless) search for a complete solution of the problem.

2.5 Invariants to Prove Non-termination

In this work, we will present an algorithm to detect non-termination. The idea of this algorithm is based on invariants. Invariants are formulae which describe a set of states. In particular, the set should have the following properties.

1. At least one state of the set is reachable in the program's execution.

2. The set is disjoint from the set of states, in which the loop condition does not hold anymore. This means the set does not contain any states in which the loop terminates.
3. Once the program's execution enters this set of states, it does not leave it anymore.

Thus, invariants are formulae which make a statement about program variables and which fulfill the following criteria. Assume that we look at a program which contains at least one (possibly infinite) loop. We can phrase the three properties equivalently as properties of invariants.

1. In the program state which the execution of the program reaches right before the loop is entered, the invariant is fulfilled by the current assignment of the program variables.
2. If the invariant holds before an arbitrary loop iteration, it also holds after the loop iteration.
3. If the invariant holds, then the loop condition is fulfilled.

If the invariant holds before the loop and implies the loop condition, the loop is entered for sure. Thus the first and the third criterion ensure that the loop is carried out at least once. If the invariant is preserved in every loop iteration and implies the loop condition, the loop is executed over and over again in case it is entered at all. This is ensured by the second and third criterion. All criteria together make sure that the loop does not terminate.

Intuitively, an invariant describes a set of states which the program's execution enters when the loop is started and never exits in the execution of the loop. An invariant can describe the set more abstract than actually necessary, as long as the set which is described does not contain states which lead to the termination of the loop.

The point is, if we find such an invariant, we have proven the non-termination of the program. From now on, we will refer to invariants which fulfill these three criteria as *non-termination invariants*.

Note, that there are other applications of invariants in reasoning about programs, too. We will have a closer look at invariants in Chapter 4 and discuss their various applications in Section 4.4.

2.6 Invariant Refinement using Failed Proofs

In this work, we will present an algorithm which generates non-termination invariants automatically. In this section, we briefly present the idea of the algorithm, which we describe in detail in Chapter 5.

In Section 3.2, we will define a logic named WHILE Dynamic Logic, in which we can reason about programs. In particular, we can use dynamic-logic formulae to express properties of programs. Such a property is for example that they do not terminate. There are theorem provers which automatically generate proofs using a proof procedure[2] which uses a calculus. Our algorithm uses such a theorem prover to prove the non-termination of programs.

Our algorithm invokes the prover to prove non-termination. During this proof procedure a non-termination invariant is required. This invariant is not provided by the procedure itself, but by our algorithm. In the first run of the procedure, the algorithm has no knowledge about the invariant and thus provides just the formula *true* as invariant candidate.

If the proof procedure is not able to close the proof, the information of the failed proof attempt is handed back to the algorithm. The algorithm inspects the failed proof and retrieves information from it to refine the invariant candidate. The refined invariant candidate is then used in another proof attempt. This process of inspection of failed proofs, invariant refinement and invocation of the proof procedure is done iteratively until one of these events occurs: a non-termination invariant is found, the invariant generator runs out of candidates or the maximum number of iterations is reached.

Thus, in the iterations of our algorithm, the invariant candidate is refined more and more in order to eventually become a non-termination invariant.

[2] The term “proof procedure” is semantically not correct, because the procedure actually undertakes the action of proving as opposed to being a part of a proof. Because the word for the action is “to prove”, it would be more logical if it was called “prove procedure”, but for some reason the term “proof procedure” is used more frequently. Although it is not logical, we obey the mainstream here and use it, too. The same applies for the term “proof obligation”, for example.

Chapter 3

Foundations

In this chapter, we explain the necessary foundations for our work. In the first section, we present the programming language WHILE. The programs of this language are the ones whose termination behavior we analysed. The following section introduces a logic, WHILE Dynamic Logic, which enables us to reason about programs. Section 3.3 describes a calculus for this logic. Section 3.4 deals with an extension of the proof procedure which enables the prover to deal with existentially quantified variables. The final section is an introduction into the theorem prover KEY, which we used in our implementation.

3.1 WHILE Programs

The first group of programs whose termination behavior we analyzed is written in a language which is a small fragment of JAVA[1]. We call a program of this language WHILE *program* and will give a formal description in Definition 3.10 in the succeeding section. The language is defined in JAVA-like syntax, but the results of our work are independent of JAVA-specific features, which means that any other imperative programming language would have served the purpose of our analysis as well.

3.1.1 Elements of WHILE Programs

The following syntactic elements of JAVA are allowed in WHILE programs.

- Variables of the primitive datatype `int` or `boolean`.
- The literals ..., `-2`, `-1`, `0`, `1`, `2`, ... and `NaN` of type `int` and `true` and `false` of type `boolean`.
- The usual operations on integers, which are multiplication `*`, addition `+`, subtraction `-`, integer division `/`, the modulo operation `%` and unary negation `-`.
- The usual operations on boolean values, which are the and-operation `&&`, or-operation `||` and the negation `!`.

[1] See [GJSB05] for the specification of JAVA.

```
collatz(int i) {
  while (i > 1) {
    if (i % 2 == 0) {
      i = i/2;
    } else {
      i = 3*i+1;
    }
  }
}
```

Figure 3.1: The Collatz program. This is one of the famous $3x+1$ problems, which were named after the German mathematicion Lothar Collatz. It is still unknown if this program terminates for all inputs.

- The equality operator `=` for both integers and boolean values and inequality operators `<`, `<=`, `>` and `>=` for integers.
- Assignments to variables. By definition of the While language, no expressions with side-effects are allowed, which leads to assignments which have only effect on the variable on the left side of the assignment. An example for such a side-effect-free assignment is:

  ```
  x = y*2 + 5;
  ```

- The conditional statement `if-then-else`, where the condition has to be a boolean expression[2] and the programs of the two branches are valid While programs.
- The loop statement `while`, where the loop condition is a boolean expression[2] and the program in the loop body is a valid While program.

For a formal definition of the language see Definition 3.10. An example of a While-program is given in Figure 3.1. Note, that we include the program name with the input parameters in the description of the examples, although these are not actually included in the While program's syntax. We do this to be able to refer to the programs in formulae without quoting the complete program every time.

3.1.2 Comparison of While and Java Programs

The following elements of Java are syntactically not allowed[3] in While programs, but they do not actually restrict the set of programs to be analyzed, because each program which uses those elements can be transformed into a program complying to the definition of While programs.

[2] Because there are no operators with side-effects allowed, any valid boolean expression will be side-effect-free.

[3] Although we exclude these elements from the While language, the examples which we show in this document might contain some of them. The reason is that the used theorem prover is able to handle them, but for the definition of the logic it is more convenient to have as few elements as possible.

- Conditions and assignments with side effects. Those can be decomposed in simple assignments to additional variables each having only effect on the assigned variable.
- Other kinds of loops: `for`-loops, `do-while`-loops, `repeat-until`-loops or `do-unless`-loops can be transformed into `while` loops using additional variables.
- `break` and `continue` statements. Loops containing `break` and `continue` can be transformed into `while` loops using additional variables. This syntactical restriction simply ensures that the only exit point from a loop is the falsification of the loop condition.
- The increase and decrease operators for integers, for example `i++`. These operations can be simulated by the usual addition and subtraction of integers.

We do not include functions or objects (so neither methods of objects) into the While language and in particular we exclude recursive functions or methods, because their termination behavior is not in the scope of this thesis (Section 2.3).

Feature restrictions of the While language like the use of objects, arrays, functions are made to prevent the language from becoming too complex to be handled. But those features (except for recursive methods and functions) are allowed in the Heap programming language, which we deal with in Chapter 8.

3.2 While Dynamic Logic

While Dynamic Logic (While DL) is a version of dynamic logic for While programs. Formulae in dynamic logic state properties of programs. The logic is based on typed first-order logic, which is a variant of first-order logic that imposes a type system on its elements. For instance, a function symbol is not only assigned the number of its inputs, but also the types of them as well as an output type.

Our version of dynamic logic comes with a syntactic feature, named updates, which is not common in definitions of dynamic logic of traditional textbooks[4]. Updates are a mechanism to save a program state during a symbolic execution. We will talk in detail about updates in Section 3.2.2, where we explain the semantics of them, and about symbolic execution in Section 3.3.1, where we present calculus rules for it.

3.2.1 Syntax of While DL

We define the elements of While DL here. Those elements are types, updates, programs, terms and formulae, of which the latter two are defined similar to traditional first-order logic[5].

[4] See [HKT00] for an example of a description of traditional dynamic logic.

[5] See [Fit96] for an introduction into first-order logic.

Types We designate a set of symbols as types. Intuitively, types in WHILE DL are similar to types in a programming language. In JAVA for instance, variables and expressions have types. The set of types for WHILE DL is defined as follows.

Definition 3.1 (WHILE DL Types)**.** We define $\mathcal{T}_{\text{WHILE}}$ as the set of types

$$\mathcal{T}_{\text{WHILE}} = \{\bot, \texttt{int}, \texttt{boolean}, \top\}$$

where $\bot$ is called the *empty type* and $\top$ the *universal type.*
There is a subtype relation $\leqslant$, which is defined upon $\mathcal{T}_{\text{WHILE}}$ as follows:

$$\begin{aligned} \bot \leqslant A \leqslant \top &\text{ for all } A \in \mathcal{T}_{\text{WHILE}} \\ \texttt{int} &\nleqslant \texttt{boolean} \\ \texttt{boolean} &\nleqslant \texttt{int} \end{aligned}$$

For convenience reasons, we leave out the subscript WHILE of $\mathcal{T}_{\text{WHILE}}$, if it is clear that we are talking about this set of types. This will be the case in all chapters from this one to Chapter 7.

The types $\bot$ and $\top$ and the subtype relation are the forerunners of a more general type system. The existence of $\bot$ and $\top$ ensures that $\mathcal{T}$ forms a lattice, which means there is a greatest common sub- and smallest common supertype for all pairs of types. Later, in Chapter 8, we will extend the notion of types to the more general system, but for dealing with WHILE programs this one is sufficient. We chose to keep the system as small as possible to not overwhelm the reader with (so far) unnecessary details.

Signatures Traditional textbooks about first-order logic[6] use signatures to define the number of inputs of function and predicate symbols. Signatures in our definition define not only the number of inputs, but also their types. Furthermore, the signature defines types also for variables, the output of function symbols and the inputs of predicate symbols.

We formally describe a signature in the following definition[7]. There, a signature resembles the signature of a method in JAVA, which also specifies the number and types of the input parameters and the type of the output parameter. Because we intend to reason about programs, it is extremely useful to include the type information for variables, functions and predicates in the definition of the logic.

The type information in our signature is given by the typing function α, which assigns types to two kinds of variables, predicate inputs, function inputs and function outputs.

Definition 3.2 (WHILE DL Signature)**.** A WHILE *DL signature* for $\mathcal{T}_{\text{WHILE}}$ is a tuple $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ consisting of

- a set $\mathcal{V}_l$ of logical variables,
- a set $\mathcal{V}_p$ of program variables,

[6] See [Fit96] for an introduction into first-order logic.
[7] This definition is loosely inspired by [BHS07], Definition 2.8.

- a set $\mathcal{F}$ of function symbols,
- a set $\mathcal{P}$ of predicate symbols, and
- a typing function α.

α has the following properties:

- $\alpha(v) \in \mathcal{T}$ for all $v \in \mathcal{V}_l \cup \mathcal{V}_r$,
- $\alpha(f) \in \mathcal{T}^n \times \mathcal{T}$ for all $f \in \mathcal{F}$ of arity n, and
- $\alpha(p) \in \mathcal{T}^n$ for all $p \in \mathcal{P}$ of arity n.

The set of function symbols $\mathcal{F}$ contains these elements:

- The binary integer operations $+, -, *, /, \% \in \mathcal{F}$.
- The unary integer operation $- \in \mathcal{F}$.
- The integer literals $\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \cdots \in \mathcal{F}$ and $\texttt{NaN} \in \mathcal{F}$[8].
- The boolean literals $\texttt{true}, \texttt{false} \in \mathcal{F}$.
- Infinitely many function symbols of each type and all arities and combinations of input types[9].
- The binary boolean operations `&&` and `||` and the unary operation `!`.

The set of rigid predicate symbols contains only these elements:

- The binary integer comparison operations $<, \leqslant, >, \geqslant \in \mathcal{P}$.
- The binary comparison operation `=` for the type $\top$[10].

We use the following notations:

- $v : A$ for $\alpha(v) = A$ for $v \in \mathcal{V}_l \cup \mathcal{V}_p$,
- $f : A_1, \ldots, A_n \rightarrow A$ for $\alpha(f) = ((A_1, \ldots, A_n), A)$ and $f \in \mathcal{F}$, and
- $p : A_1, \ldots, A_n$ for $\alpha(p) = (A_1, \ldots, A_n)$ and $p \in \mathcal{P}$.

We distinguish logical variables from program variables, because it simplifies the calculus. For example, we want to restrict the application of substitutions to logical variables, because we do not appreciate any substitutions in programs[11].

Example 3.3 (WHILE DL Signature)**.** As an example, have a look at the following signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P})$ with

$$\mathcal{V}_l = \{k, l\} \quad \text{and} \quad \mathcal{V}_p = \{\texttt{a}, \texttt{b}, \texttt{e}\}$$

[8] The symbol `NaN` is included in $\mathcal{F}$ to handle division by zero; see Section 3.2.2.

[9] This is necessary, because the calculus rules allRight and exLeft introduce function symbols which have not been used so far. Because of the metavariables (see Section 3.4.1), these function symbols need to have input terms, because they indicate which metavariables the symbols is dependent of. This is described in detail in Section 3.3.1.

[10] Because this is a supertype of all types, the predicate can be applied on all other types in $\mathcal{T}$ as well.

[11] Those substitutions would violate some essential properties of substitions, for example the substitution lemma. See [Fit96], Section 5.2 for more details on substitution.

and $\mathcal{F}$ and $\mathcal{P}$ as defined in the preceding definition. α assigns the following types:

$$\begin{array}{lllllllll} \texttt{a} & : & \texttt{int} & k & : & \texttt{boolean} & \texttt{b} & : & \texttt{int} \\ l & : & \texttt{int} & \texttt{e} & : & \texttt{boolean} & & & \end{array}$$

Updates Updates are a part of formulae which contain information about program states. To be precise, they describe the difference from a starting state to the current state of a program during an execution.

Updates look like common assignment statements in programs. On the left side of an update there is a program variable which is assigned the term on the right side of the update. Updates can be performed in parallel, which is syntactically expressed by separating multiple updates by the symbol $||$.

Example 3.4 (While DL Updates)**.** Have a look at the following examples of updates

$$\begin{array}{lll} \texttt{a} := 2 & \texttt{b} := 4 + l & \texttt{e} := k \\ \texttt{e} :=!\texttt{e} & \texttt{a} := \texttt{a} * (-1) \mid\mid \texttt{e} := k \;\texttt{\&\&}\; \texttt{e} & \end{array}$$

The following definition[12] describes updates in detail. Updates are not a standard element of dynamic logic, but happen to be an intuitive way to deal with programs and their states. Beckert et al. introduce them in [BHS07] for the Java Card Dynamic Logic.

Note, that we define updates using the notion of terms and vice versa. We chose this mutually inductive definition, because this was the simplest way to introduce them. We preferred this approach to give the reader a chance to grab the concept in an intuitive way.

Definition 3.5 (While DL Updates)**.** Let $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$, a While DL signature for the type set $\mathcal{T}$ be given, then the set $\mathfrak{U}$ of *syntactic updates* is inductively defined as the least set such that:

- $(v := t) \in \mathfrak{U}$ for all $v \in \mathcal{V}_p$ of type A and $t \in \mathfrak{T}_A$[13] (variable update), and
- $(u_1 \mid\mid u_2) \in \mathfrak{U}$ for all $u_1, u_2 \in \mathfrak{U}$ (parallel update).

Terms Terms in a logic are similar to expressions in a programming language. We form terms by applying function symbols to logical or program variables, constants[14] and other terms. Another way of forming terms is to precede another term with an update.

The formal definition[15] of terms does not differ much from the definition in traditional textbooks[16] except that the types of the entities have to comply to the signature and that updates can be part of terms.

[12] The definition is a fragment of Definition 3.8 of [BHS07].

[13] $\mathfrak{T}_A$ is the set of terms of type A and defined in Definition 3.6.

[14] Constants are technically nullary function symbols.

[15] Taken from [BHS07], Definition 2.15.

[16] See for example [Fit96], Definition 5.1.2.

Definition 3.6 (While DL Terms)**.** Given a While DL signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ for the set of types $\mathcal{T}$, the system $\{\mathfrak{T}_A\}_{A \in \mathcal{T}}$ of sets of *terms of type* A is inductively defined as the least system of sets such that:

- $x \in \mathfrak{T}_A$ for all variables $x \in \mathcal{V}_l \cup \mathcal{V}_p$ with $x : A$,
- $f(t_1, \ldots, t_n) \in \mathfrak{T}_a$ for all function symbols $f : A_1, \ldots, A_n \to A$ in $\mathcal{F}$ and terms $t_i \in \mathfrak{T}_{C_i}$ and $C_i \leqslant A_i$ for $1 \leqslant i \leqslant n$, and
- $\{u\}t \in \mathfrak{T}_A$ for all updates $u \in \mathfrak{U}$ and all terms $t \in \mathfrak{T}_A$.

Example 3.7 (While DL Terms)**.** The following words are examples for valid and invalid terms of the signature defined in Example 3.3.

1. $\mathtt{a} \in \mathfrak{T}_{\mathtt{int}}$
2. $l * 2 \in \mathfrak{T}_{\mathtt{int}}$
3. $\mathtt{a} + \mathtt{b} + 4 \in \mathfrak{T}_{\mathtt{int}}$
4. $\mathtt{e}\ \mathtt{\&\&}\ k \in \mathfrak{T}_{\mathtt{boolean}}$
5. $\{\mathtt{b} := 5\}(\mathtt{b} * \mathtt{b}) \in \mathfrak{T}_{\mathtt{int}}$
6. $\mathtt{a} * k \notin \mathfrak{T}$
7. $\mathtt{max}(\mathtt{a}, \mathtt{b}) \notin \mathfrak{T}$

Example 6 is not a valid term, because the variable k is of the wrong type to be an input variable to the function $*$. Example 7 is not a valid term, because the signature does not contain a function `max`.

We are going to identify two subsets of terms, ground terms and rigid terms. Note, that, according to our definition, ground terms are allowed to contain variables, but only program variables and no logical variables.

Definition 3.8 (Ground Terms)**.** A term which contains no logical variables and no updates is called *ground term.*

The terms $\mathtt{a}$ and $\mathtt{a} + \mathtt{b} + 4$ of Example 3.7 are ground terms, but terms $l * 2$ and $\mathtt{e}\ \mathtt{\&\&}\ k$ are not, because they contain logical variables. The term $\{\mathtt{b} := 5\}(\mathtt{b} * \mathtt{b})$ is not a ground term, because it contains an update.

We will define a calculus for While DL in Section 3.3. Some of the rules in this calculus distinguish between terms that can change their meaning from one program state to another and those that cannot. The latter set of terms is called rigid terms and we describe it in the following definition[17].

Definition 3.9 (Rigid Terms)**.** A While term t is *rigid,*

- if $t = x$ and $x \in \mathcal{V}_l$,
- if $t = f(t_1, \ldots, t_n)$, $f \in \mathcal{F}$ and the subterms t_i are rigid $(1 \leqslant i \leqslant n)$, or
- if $t = \{u\}s$ and s is rigid.

Of the terms in Example 3.7 only the term $l * 2$ is rigid, because all others contain program variables.

[17] Taken from [BHS07], Definition 3.32.

Programs Formulae in WHILE DL state properties of programs. Therefore programs are a syntactic part of the logic. The following definition is the formal description of the WHILE language, which we introduced in Section 3.1.

Definition 3.10 (WHILE Programs)**.** The set of WHILE programs $\mathfrak{P}$ is inductively defined by

- $v = t; \in \mathfrak{P}$, where $v \in \mathcal{V}_p$ with $v : A$ and $t \in \mathfrak{T}_C$ a ground term with $C \in \{\texttt{int}, \texttt{boolean}\}$ and $C \leqslant A$;
- $p_1 p_2 \in \mathfrak{P}$, if $p_1, p_2 \in \mathfrak{P}$;
- $\texttt{if (} c \texttt{) \{ } p_1 \texttt{ \} else \{ } p_2 \texttt{ \}} \in \mathfrak{P}$ and $\texttt{if (} c \texttt{) \{ } p_1 \texttt{ \}} \in \mathfrak{P}$, if $c \in \mathfrak{T}_{\texttt{boolean}}$ a ground term and $p_1, p_2 \in \mathfrak{P}$; and
- $\texttt{while (} c \texttt{) \{ } p \texttt{ \}} \in \mathfrak{P}$, if $c \in \mathfrak{T}_{\texttt{boolean}}$ ground term and $p \in \mathfrak{P}$.

We gave an example for a WHILE program in Figure 3.1.

Formulae A formula in a logic resembles a boolean expression in a programming language. In traditional first-order logic, we form formulae only by applying predicates to terms or compose them from other formulae using logical connectives and quantifiers. In dynamic logic, we can precede a formula by a program. The intended semantics, which we will define in Section 3.29, are that the formulae states a property about the resulting state of the program. Our specific logic provides one more way to form a formula, which is to prefix an update to a subformula.

Although a formula is similar to boolean valued terms or boolean expressions in a programming language, we distinguish between formulae and terms or expressions. Formulae can be composed by more elements than terms, for example quantifiers and logical connectives. Furthermore, boolean expressions in programming languages can have side effects on variables[18], where formulae do not change the value of (logical) variables. The formal definition[19] of formulae follows.

Definition 3.11 (WHILE DL Formulae)**.** Let a signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ for the type set $\mathcal{T}$ and a WHILE program p be given. Then the set $\mathfrak{F}$ of WHILE DL formulae is inductively defined as the least set such that

- $r(t_1, \ldots, t_n) \in \mathfrak{F}$ for all predicate symbols $r : A_1, \ldots, A_n \in \mathcal{P}$ and terms $t_i \in \mathfrak{T}_{C_i}$ with $C_i \leqslant A_i$ for $1 \leqslant i \leqslant n$,
- $\mathit{true}, \mathit{false} \in \mathfrak{F}$,
- $\neg\varphi, \varphi \vee \psi, \varphi \wedge \psi, \varphi \rightarrow \psi, \varphi \leftrightarrow \psi \in \mathfrak{F}$ for all $\varphi, \psi \in \mathfrak{F}$.
- $\forall x\ \varphi, \exists x\ \varphi \in \mathfrak{F}$ for all $\varphi \in \mathfrak{F}$ and all variables $x \in \mathcal{V}_l$,
- $\{u\}\varphi \in \mathfrak{F}$ for all $\varphi \in \mathfrak{F}$ and $u \in \mathfrak{U}$, and
- $\langle p \rangle \varphi, [p]\varphi \in \mathfrak{F}$ for all $\varphi \in \mathfrak{F}$ and $p \in \mathfrak{P}$.

[18] Although, in the WHILE language as we defined it in Section 3.1, boolean expressions cannot have side effects, but in general there are programming languages in which they can.

[19] This definition is inspired by Definition 3.14 of [BHS07].

The symbol $\forall$ in the formula $\forall x\ \varphi$ is called *universal quantifier* and the symbol $\exists$ in the formula $\exists x\ \varphi$ is called *existential quantifier.* They *bind* the variable x in their *scope*, which is the subformula φ. We denote variables that are not bound as *free* and define them formally in Definition 3.14[20]. *Closed* formulae do not contain free variables. Note, that we can only quantify over logical variables, but not over program variables[21].

Example 3.12 (WHILE DL Formulae)**.** The following formulae are proper WHILE DL formulae using the signature of Example 3.3:

1. $\mathtt{a} + \mathtt{b} = \mathtt{b} + 5$
2. $\{\mathtt{a} := 10\}\ \mathtt{a} > 20$
3. $[\mathtt{if\ (}\ a > b\ \mathtt{)}\ \{\ a = b\ \}\ \mathtt{else}\ \{\ b = a\ \}](a = b)$
4. $\{\mathtt{e} := \mathtt{false}\}[\mathtt{e\ =\ (!e);}](\mathtt{e} = \mathtt{true})$
5. $\exists x\ (l = x * x)$
6. $\forall x\ \{\mathtt{b} := x\}[\mathtt{b\ =\ b\ +\ 5}](b > x)$

As for terms, we distinguish between formulae which can evaluate differently in different program states and those which cannot. The latter are called rigid formulae and are described in the following definition[22].

Definition 3.13 (Rigid Formulae)**.** A WHILE DL formula φ is *rigid*,

- if $\varphi = p(t_1, \ldots, t_n), p \in \mathcal{P}$ and the terms t_i are rigid with $1 \leqslant i \leqslant n$,
- if $\varphi = \mathit{true}$ or $\varphi = \mathit{false}$,
- if $\varphi = \neg\psi$ and ψ is rigid,
- if $\varphi = \psi_1 \vee \psi_2, \varphi = \psi_1 \wedge \psi_2$, or $\varphi = \psi_1 \rightarrow \psi_2$, and ψ_1, ψ_2 are rigid,
- if $\varphi = \forall x\ \psi$ or $\psi = \exists x\psi$, and ψ is rigid, or
- if $\varphi = \{u\}\psi$ and ψ is rigid.

Of the formulae in Example 3.12 only $\exists x\ (l = x * x)$ is rigid, because all other ones contain program variables.

Definition 3.14 (Free Variables in WHILE DL)**.** We define the set $\mathrm{fv}(u)$ of free variables of an update u by:

- $\mathrm{fv}(v := t) = \mathrm{fv}(t)$ and
- $\mathrm{fv}(u_1||u_2) = \mathrm{fv}(u_1) \cup \mathrm{fv}(u_2)$.

We define $\mathrm{fv}(t)$, the set of free variables of a term t, by

[20] Taken from [BHS07], Definition 2.17 and 3.14.

[21] This is a technical decision, because it simplifies the calculus. It would be possible to define a logic where we can quantify over program variables as well. We chose this way to preserve classical properties of substition. For details on substitution, see [Fit96], Section 5.2.

[22] Taken from [BHS07], Definition 3.37.

- $\text{fv}(v) = \{v\}$ for $v \in \mathcal{V}_l$,
- $\text{fv}(x) = \emptyset$ for $x \in \mathcal{V}_p$,
- $\text{fv}(f(t_1, \dots, t_n)) = \bigcup_{i=1,\dots,n} \text{fv}(t_i)$ for $f \in \mathcal{F}$, and
- $\text{fv}(\{u\}t) = \text{fv}(u) \cup \text{fv}(t)$ for $t \in \mathfrak{T}$ and $\{u\} \in \mathfrak{U}$.

We define $\text{fv}(\varphi)$, the set of free variables of a formula φ, by

- $\text{fv}(p(t_1, \dots, t_n)) = \bigcup_{i=1,\dots,n} \text{fv}(t_i)$ for $p \in \mathcal{P}$,
- $\text{fv}(\mathit{true}) = \text{fv}(\mathit{false}) = \emptyset$,
- $\text{fv}(\neg\varphi) = \text{fv}(\varphi)$,
- $\text{fv}(\varphi \wedge \psi) = \text{fv}(\varphi \vee \psi) = \text{fv}(\varphi \rightarrow \psi) = \text{fv}(\varphi) \cup \text{fv}(\psi)$,
- $\text{fv}(\forall x\ \varphi) = \text{fv}(\exists x\ \varphi) = \text{fv}(\varphi) \backslash \{x\}$,
- $\text{fv}(\{u\}\varphi) = \text{fv}(u) \cup \text{fv}(\varphi)$ for a formula φ, and
- $\text{fv}(\langle p \rangle\varphi) = \text{fv}([p]\varphi) = \text{fv}(\varphi)$ for a formula φ.

Of the formulae in Example 3.12 only the formula $\exists x\ (l = x * x)$ has a free variable, namely l. All other formulae have no free variables, because their variables are either program variables or bound by a quantifier.

3.2.2 Semantics of While

The semantics of While DL captures the intuitive meaning of programs. A state of a program is completely specified by an assignment of program variables. A program can be considered as a function on program states, which simply means it converts one state to another by performing its action.

We define the semantics of all elements of While DL step by step beginning with models. Models[23] form the basis of the semantics by providing a domain and an interpretation. A domain is a set of elements of each available type. The elements of the domain are the values which can be assigned to variables. An interpretation is a mapping of function and predicate symbols to actual functions and predicates.

Definition 3.15 (While DL Model)**.** Given the type set $\mathcal{T}_{\text{While}}$ and a While DL signature, a While *model* is the triple $\mathcal{M}_{\text{While}} = (\mathcal{D}_{\text{While}}, \delta_{\text{While}}, \mathcal{I}_{\text{While}})$ with

- the *domain* $\mathcal{D}_{\text{While}}$, such that

$$\mathcal{D}_{\text{While}} = \mathcal{D}_{\texttt{int}} \cup \mathcal{D}_{\texttt{boolean}}$$

  with

$$\mathcal{D}_{\texttt{int}} = \{\dots, -2, -1, 0, 1, 2, \dots\} \quad \text{and} \quad \mathcal{D}_{\texttt{boolean}} = \{\mathit{true}, \mathit{false}\};$$

[23] The following definition is inspired by Definition 2.20 of [BHS07].

- the type function $\delta_{\textsc{While}} : \mathcal{D}_{\textsc{While}} \to \mathcal{T}_{\textsc{While}}$ with

$$\begin{array}{ll} \delta_{\textsc{While}}(i) = \texttt{int} & \text{for } i \in \mathcal{D}_{\texttt{int}} \text{ and} \\ \delta_{\textsc{While}}(b) = \texttt{boolean} & \text{for } b \in \mathcal{D}_{\texttt{boolean}}; \end{array}$$

- and the *interpretation* $\mathcal{I}_{\textsc{While}}$, which maps each function symbol $f : A_1, \ldots, A_n \to A \in \mathcal{F}$ to a function

$$\mathcal{I}_{\textsc{While}}(f) : \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n} \to \mathcal{D}_A,$$

  and each predicate symbol $p : A_1, \ldots, A_n \in \mathcal{P}$ to a subset

$$\mathcal{I}_{\textsc{While}}(p) \subseteq \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n},$$

  with $A_1, \ldots, A_n, A \in \mathcal{T}$.

$\mathcal{I}_{\textsc{While}}(f)$ maps the literals $\ldots, \texttt{-2}, \texttt{-1}, \texttt{0}, \texttt{1}, \texttt{2}, \ldots$ and true, false to their intuitive counterparts in $\mathcal{D}_{\texttt{int}}$ and $\mathcal{D}_{\texttt{boolean}}$. The symbol NaN is mapped to an integer which is not specified any further here. Besides that, $\mathcal{I}_{\textsc{While}}(f)$ relates the intuitive algebraic and boolean predicates to the predicates $<, \leqslant, >, \geqslant, =,$ &&, ||, which we defined in Definition 3.2.

For the binary symbols $+, -, * \in \mathcal{F}$ and the unary $- \in \mathcal{F}$, we demand that $\mathcal{I}_{\textsc{While}}(f)$ yields the intuitive algebraic operations. The modulo operator % and the division operator / are assigned the Euclidean modulo and division operators as they are defined in [Bou92], Section 2.4.

For convenience, we omit the subscript WHILE of symbols, if it is clear about which version of the symbol we are talking.

Note 3.16 (On Modulo and Division Operators)**.** The definition of the modulo and division operators varies among different programming languages. The one which makes most sense from the mathematical viewpoint is the one we chose, the Euclidean division and modular operators from [Bou92], Section 2.4. We quote the definition here to clarify all details.

Definition 3.17 (Euclidean Division and Modulo Operation)**.** With a, b of type int, we define

$$a \mathbin{/} b = q \quad \text{and} \quad a \mathbin{\%} b = r$$

where q, r are of type int and represent the corresponding entities in Euclid's theorem, Theorem 3.18.

Theorem 3.18 (Euclid's Theorem)**.** For any real[24] numbers $a', b' \in \mathbb{R}$ with $b' \neq 0$, there exists a unique pair of numbers $q, r \in \mathbb{R}$ satisfying the following conditions.

$$q \in \mathbb{Z}, \quad a' = b' \cdot q + r \quad \text{and} \quad 0 \leqslant r < |b'|$$

[24] And thus also for any integer numbers.

The interpretation of the modulo and the division operator differs from the operators as they are implemented in JAVA. We chose this interpretation, because we want to present our results as independent of a particular programming language as possible. However, it is possible to construct examples where the difference in the interpretations can make a difference in the termination behavior of the program, but this does not affect the general idea of the method which we developed in this work.

To evaluate a formula or a term, we need to assign values to variables. Therefore we define a variable assignment to be a mapping between the set of variables and the domain, which contains the values. The following two definitions[25] define this mapping for the two types of variables, program and logical ones.

Definition 3.19 (Logical Variable Assignment)**.** Given a model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, a *logical variable assignment* is a function $\beta : \mathcal{V}_l \to \mathcal{D}$, such that

$$\beta(x) \in \mathcal{D}_A \text{ for all } x : A \in \mathcal{V}_l.$$

We also define the modification β_x^d of a variable assignment β for any variable $x : A$ and any domain element $d \in \mathcal{D}_A$ by:

$$\beta_x^d(y) := \begin{cases} d & \text{if } y = x \\ \beta(y) & \text{otherwise.} \end{cases}$$

Example 3.20 (Logical Variable Assignment)**.** An example for a logical variable assignment for the example signature which we defined in Example 3.3 is:

$$\beta(l) = 2 \text{ and } \beta(k) = \mathit{false}.$$

A modification of β is β_l^4 which is defined by

$$\beta_l^4(l) = 4 \text{ and } \beta_l^4(k) = \beta(k) = \mathit{false}.$$

Definition 3.21 (Program Variable Assignment)**.** Given a model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, a *program variable assignment* is a function $\gamma : \mathcal{V}_p \to \mathcal{D}$, such that

$$\gamma(v) \in \mathcal{D}_A \text{ for all } v \in \mathcal{V}_p, v : A, A \in \mathcal{T}$$

Example 3.22 (Program Variable Assignment)**.** An example for a program variable assignment for the example signature defined in Example 3.3 is:

$$\gamma(\mathtt{a}) = 2,\ \gamma(\mathtt{b}) = -5 \ \text{ and } \ \gamma(\mathtt{e}) = \mathtt{true}$$

A program variable assignment is a state of a program. Note, that such an assignment is always a function from the set of program variables into the domain. In Definition 3.25, we will define the semantics of programs as functions from states to states. To define the semantics of *non-terminating* programs as well, we need

[25] The definition of logical variable assignments is taken from [BHS07], Definition 2.23

another state, namely one which is not represented by a variable assignment. This additional state serves as "output" state of non-terminating programs, although of course they literally do not have any final state. Because we need the notion of a state space with such a state, we invent a new state and introduce an extended version of the state space.

Definition 3.23 (Program State and State Space)**.** Given a model $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$, the set $\mathcal{S}^{\mathcal{M}}$ of program states is defined as

$$\mathcal{S}^{\mathcal{M}} = \{\gamma \mid \gamma : \mathcal{V}_p \to \mathcal{D}\}$$

We define an additional state, named $s_\infty \notin \mathcal{S}^{\mathcal{M}}$. We define the state space $\mathcal{S}^{\mathcal{M}}_\infty$ which is $\mathcal{S}^{\mathcal{M}}$ extended by s_∞:

$$\mathcal{S}^{\mathcal{M}}_\infty = \mathcal{S}^{\mathcal{M}} \cup \{s_\infty\}$$

With a domain, variable assignments and semantics of function symbols, we can now give terms a meaning. We calculate a value for a term from the values which are assigned to the variables by the application of the functions which are involved in the term.

Note, we use the semantics of updates $[\![u]\!]^{\mathcal{M},\beta}(\gamma)$ in the definition of terms and vice versa. This is due to the fact that we already defined the syntax like that, because it is the most intuitive way to understand the relation between those entities. The definition of the overriding operator $\oplus$ and of the semantics of updates follows in Definition 3.28.

Definition 3.24 (Semantics of Terms)**.** Let $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ be a model, β a logical variable assignment and γ a program variable assignment. We inductively define the valuation function $\mathrm{val}_{\mathcal{M},\beta,\gamma}$ for terms as

- $\mathrm{val}_{M,\beta,\gamma}(v) = \beta(v)$ for any variable $v \in \mathcal{V}_l$,
- $\mathrm{val}_{M,\beta,\gamma}(w) = \gamma(w)$ for any variable $w \in \mathcal{V}_p$,
- $\mathrm{val}_{M,\beta,\gamma}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(\mathrm{val}_{M,\beta,\gamma}(t_1), \ldots, \mathrm{val}_{M,\beta,\gamma}(t_n))$ for every $f \in \mathcal{F}$ and $t_i \in \mathfrak{T}$, and
- $\mathrm{val}_{M,\beta,\gamma}(\{u\}t) = \mathrm{val}_{\mathcal{M},\beta,\gamma \oplus [\![u]\!]^{\mathcal{M},\beta}(\gamma)}(t)$ for $u \in \mathfrak{U}$ and $t \in \mathfrak{T}$.

We define σ to be the function which maps each term to the type of its valuation.

Programs can be considered as partial functions on the set of states. A program maps an input state to a result state, which is reached if we execute the program with the input state as start state. This notion is captured in the following definition[26] of the semantics of programs.

Definition 3.25 (Semantics of Programs)**.** Given a $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, a program is a function:

$$[\![p]\!]^{\mathcal{M}} : \mathcal{S}^{\mathcal{M}}_\infty \to \mathcal{S}^{\mathcal{M}}_\infty.$$

[26] This definition is inspired by [HPRW06].

Concerning the state s_∞, all programs have the same semantics:

$$[\![p]\!]^{\mathcal{M}}(s_\infty) = s_\infty$$

The semantics of the program elements of Definition 3.10 follow for all other states $\gamma \neq s_\infty$[27].

- Assignments:
$$[\![v = t]\!]^{\mathcal{M}}(\gamma)(x) = \begin{cases} \mathrm{val}_{\mathcal{M},\gamma}(t) & \text{if } x = v \\ \gamma(x) & \text{otherwise,} \end{cases}$$
- Composition:
$$[\![p_1 p_2]\!]^{\mathcal{M}} \;:=\; [\![p_2]\!]^{\mathcal{M}} \circ [\![p_1]\!]^{\mathcal{M}},$$
where $\circ$ denotes the composition of functions.
- Loops:
$$[\![\texttt{while (} c \texttt{) \{ } p \texttt{ \}}]\!]^{\mathcal{M}} \;:=\; \lim_{i\to\infty} w_i,$$
where w_i is defined as
$$w_i : \mathcal{S}_\infty^{\mathcal{M}} \to \mathcal{S}_\infty^{\mathcal{M}},$$
and
$$\begin{array}{rcl} w_0(\gamma) & = & s_\infty \\ w_{i+1}(\gamma) & = & \begin{cases} w_i([\![p]\!]^{\mathcal{M}}(\gamma)) & \text{for } \mathrm{val}_{\mathcal{M},\gamma}(c) = \mathit{true} \\ \gamma & \text{otherwise.} \end{cases} \end{array}$$
and
$$\begin{array}{c} \lim_{i\to\infty} w_i(x) = y \quad \text{if} \\ w_j(x) = y \text{ for almost all } j \in \mathbb{N}. \end{array}$$
- Conditionals:
$$[\![\texttt{if (} c \texttt{) \{ } p_1 \texttt{ \} else \{ } p_2 \texttt{ \}}]\!]^{\mathcal{M}}(\gamma) \;:=\; \begin{cases} [\![p_1]\!]^{\mathcal{M}}(\gamma) & \text{if } \mathrm{val}_{\mathcal{M},\gamma}(c) = \mathit{true} \\ [\![p_2]\!]^{\mathcal{M}}(\gamma) & \text{otherwise} \end{cases}$$

From the definition of the semantics of the composition of programs we can conclude that the composition is associative. That means the programs $(p_1p_2)p_3$ and $p_1(p_2p_3)$ are identical.

We state a specific example to make the semantics of programs clearer.

Example 3.26 (Semantics of Programs)**.** Let p be the program StateSpace in Figure 3.2. According to Definition 3.25, the semantics is the function

$$[\![p]\!] : (\mathcal{D}_{\texttt{int}} \times \mathcal{D}_{\texttt{int}}) \cup \{s_\infty\} \to (\mathcal{D}_{\texttt{int}} \times \mathcal{D}_{\texttt{int}}) \cup \{s_\infty\} : (a,i) \mapsto [\![p]\!](a,i)$$

[27] Note, that the valuation of expressions like t in the assignment or c in the conditional in this definition depends only on the model $\mathcal{M}$ and the assignment γ of program variables and not on the assignment of logical variables, because terms which can occur in programs are ground terms only (see Definition 3.10).

```
statespace(int i, int a) {
  while (0 <= i && i < 2) {
    a = 2*a;
    i = i+1;
  }
}
```

Figure 3.2: StateSpace
A simple program with two input variables.

with

(a,i)	$\llbracket p \rrbracket(a,i)$
s_∞	s_∞
$(a,0)$	$(a*4,2)$
$(a,1)$	$(a*2,2)$
(a,i)	(a,i) for $i<0$ or $i \geqslant 2$

Updates are similar to programs, but in contrast to programs they are not functions between program states but from program states to partial variable assignments. Thus, the application of an update does not yield a full description of a program state, but a description of the difference from the program state before the update to the state after the update.

Semantically, updates are functions from the set of program states to the set of partial functions from the program variables to the domain. The parallel application of updates is represented by the different partial functions which override each other. If a parallel update is applied and different updates of the parallel one contradict each other, always the rightmost one wins. To define these semantics, we define[28] an overriding operator $\oplus$, which captures exactly this behavior.

Definition 3.27 (Overriding Operator)**.** Given a set M, for two (partial or total) functions $f, g : M \to M$, we define

$$f \oplus g := \{(a \mapsto b) \in f \mid \text{ for all } c : (a \mapsto c) \notin g\} \cup g,$$

which means g overrides f but leaves f unchanged at points where g is not defined.

With the notion of the overriding operator, we can now define[28] the semantics of updates.

Definition 3.28 (Semantics of Updates)**.** Updates $u \in \mathfrak{U}$ are partial functions from the state space to partial variable assignments

$$\llbracket u \rrbracket^{\mathcal{M},\beta} : \mathcal{S}^{\mathcal{D}} \to (\mathcal{V}_p \twoheadrightarrow \mathcal{D}),$$

where $\mathcal{V}_p \twoheadrightarrow \mathcal{D}$ stands for a partial function from $\mathcal{V}_p$ to $\mathcal{D}$. The semantics for single and parallel updates are

$$\llbracket w := t \rrbracket^{\mathcal{M},\beta}(\gamma) \ := \ \{w \mapsto \mathrm{val}_{\mathcal{M},\beta,\gamma}(t)\}$$

[28] This definition is taken from [Rüm06], Section 4.

and

$$\llbracket w_1 := t_1 \mid\mid \ldots \mid\mid w_k := t_k \rrbracket^{\mathcal{M},\beta}(\gamma) \; := \\ \llbracket w_1 := t_1 \rrbracket^{\mathcal{M},\beta}(\gamma) \oplus \llbracket w_2 := t_2 \rrbracket^{\mathcal{M},\beta}(\gamma) \oplus ... \oplus \llbracket w_k := t_k \rrbracket^{\mathcal{M},\beta}(\gamma).$$

At this point, we have defined all ingredients to define the semantics of formulae[29]. In the following definition, we do not distinguish between the formula *true* and the truth value *true*. A formula is valid, if it is equivalent to the formula *true*.

Definition 3.29 (Semantics of Formulae)**.** Let $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ be a model, β a logical variable assignment and γ a program variable assignment. We inductively define the valuation function $\mathrm{val}_{\mathcal{M},\beta,\gamma}$ for formulae as follows.

- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\mathit{true}) = \mathit{true}$.
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\mathit{false}) = \mathit{false}$.
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(p(t_1, \ldots, t_n)) = \mathit{true}$ iff $(\mathrm{val}_{\mathcal{M},\beta,\gamma}(t_1), \ldots, \mathrm{val}_{\mathcal{M},\beta,\gamma}(t_n)) \in \mathcal{I}(p)$ for $p \in \mathcal{P}$.
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\neg\varphi) = \mathit{true}$ iff $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi) = \mathit{false}$.
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi \wedge \psi)$ iff $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi) = \mathit{true}$ and $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\psi) = \mathit{true}$.
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi \vee \psi)$ iff $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi) = \mathit{true}$ or $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\psi) = \mathit{true}$ (or both).
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\forall x \; \varphi) = \mathit{true}$ iff $\mathrm{val}_{\mathcal{M},\beta^d_y,\gamma}(\varphi)$ for every $d \in \mathcal{D}_A$ and $x : A$.
- $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\exists x \; \varphi) = \mathit{true}$ iff $\mathrm{val}_{\mathcal{M},\beta^d_x,\gamma}(\varphi)$ for some $d \in \mathcal{D}_A$ and $x : A$.
- $\mathrm{val}_{M,\beta,\gamma}(\{u\}\varphi) = \mathrm{val}_{\mathcal{M},\beta,\gamma\oplus\llbracket u \rrbracket^{\mathcal{M},\beta}(\gamma)}(\varphi)$ for $u \in \mathfrak{U}$ and $\varphi \in \mathfrak{F}$.
- If $p \in \mathfrak{P}$, then

$$\mathrm{val}_{M,\beta,\gamma}([p]\varphi) = \begin{cases} \mathrm{val}_{\mathcal{M},\beta,\llbracket p \rrbracket^{\mathcal{M},\beta}(\gamma)}(\varphi) & \text{for } \llbracket p \rrbracket^{\mathcal{M},\beta} \neq s_\infty \\ \mathit{true} & \text{otherwise.} \end{cases}$$

- $\mathrm{val}_{M,\beta,\gamma}(\langle p \rangle\varphi) = \mathrm{val}_{M,\beta,\gamma}(\neg[p]\neg\varphi)$.

Note 3.30 (Partial Correctness and Total Correctness)**.** When reasoning about programs, we distinguish two types of proofs, *partial correctness proofs* and *total correctness proofs*. In partial correctness proofs, the termination of a program is not mandatory. Partial correctness proofs prove properties of a program *provided that* the program terminates. In situations where it does not terminate, the proof does not say anything about the program.

In contrast, total correctness proofs require the proof of termination. Total correctness proofs can be split up into two parts[30]: the proof of the termination and the proof of the actual statement which the user intended to make about the program. If the user does not want to prove anything else besides the termination, this sole termination proof is also called a total correctness proof.

[29] This definition was inspired by Definition 2.26 of [BHS07] and the definition of the semantics of programs which contain modalities in [HPRW06], Section 3.2.

[30] This is only correct for deterministic programs. Both programming languages which we used, While and Heap are deterministic.

In dynamic logic, those two types of correctness are represented by the two types of modalities, which are provided to integrate programs into formulae. The formula $[p]\varphi$ for a program p and a formula φ says: "If the program p terminates, the formulae φ holds after the execution". In contrast, the formula $\langle p\rangle\varphi$ states: "The program p does terminate and after the execution φ holds".

Validity and Satisfiability Essential properties of formulae are validity and satisfiability. In first-order logic, those properties are defined given a model and a logical variable assignment. We interpret logical variables existentially and program variables universally, which leads to the following definition[31].

Definition 3.31 (Validity, Satisfiability, Unsatisfiability)**.** A formula φ is *valid*, iff for each model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ and each program variable assignment $\gamma \in \mathcal{S}^{\mathcal{M}}$ there is a logical variable assignment $\beta : \mathcal{V}_l \to \mathcal{D}$ such that $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi) = \mathit{true}$.

A formula φ is *satisfiable*, iff there is a model $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, program variable assignment $\gamma \in \mathcal{S}^{\mathcal{M}}$, and a logical variable assignment $\beta : \mathcal{V}_l \to \mathcal{D}$ such that $\mathrm{val}_{\mathcal{M},\beta,\gamma}(\varphi) = \mathit{true}$.

A formula φ is *unsatisfiable*, iff $\neg\exists x_1 \ldots \exists x_n\ \varphi$ is valid, where $x_1, \ldots, x_n$ are the free variables of φ.

3.3 While DL Calculus

A calculus is a formalism to derive the validity of formulae. Constructing a proof using a calculus means applying purely syntactic operations on formulae. The calculus which we present here is a derivation of a Gentzen-sequent calculus[32].

The starting point of a proof is a sequent formula, to which the formula in question is translated. Sequent formulae are a particular form of sets of formulae, which we will define soon. We apply a rule on the sequent formula, resulting in one or more new sequent formulae. We go on applying rules on the sequent formulae until we reach a set of end sequent formulae, called axioms, which are trivially valid.

The direction of the proof construction is *backward*, because we start with what we want to show and end up with what we knew before. The way to read a proof is the other way round, because naturally we start with what we know and combine these facts with each other until we reach the statement which we want to prove. So, the presented calculus is a method of *backward reasoning*.

A proof is represented by a tree whose nodes are sequent formulae. The sequent formula whose validity we want to prove is the root node. Each edge between the nodes (sequent formulae) corresponds to the application of a rule. A rule consists of one or more premises and one conclusion. The application of a rule results in as many new formulae as the rule has premises. The premises lead to different branches in the proof tree. The leafs of the branches contain in the best case the axioms. In

[31] Taken from [HPRW06], Section 4.

[32] See [Fit96], Section 6.6 for further information on Gentzen-sequent calculi.

cases where we are not able to prove the formula, we receive a set of formulae which are not axioms. These leafs are called *open goals*.

Reasons for not being able to close a proof can be either that the formula we want to prove is not valid or the set of calculus rules is insufficient for this proof.

We describe proof trees in Definition 3.33[33]. The formulae in the proof-tree nodes are sequent formulae, whose formal definition[34] follows.

Definition 3.32 (Sequent, Antecedent and Succedent)**.** A *sequent formulae* or *sequent* is a pair of sets of formulae written as

$$\varphi_1, \ldots, \varphi_m \Rightarrow \varphi_1, \ldots, \varphi_n.$$

The set of formulae φ_i on the left of the *sequent arrow* "⇒" is called the *antecedent*, the set of formulae ψ_j on the right the *succedent* of the sequent. We use capital Greek letters to denote several formulae in the antecedent or succedent of a sequent, so by

$$\Gamma, \varphi \Rightarrow \psi, \Delta$$

we mean a sequent containing φ in the antecedent, and ψ in the succedent, as well as possibly none or many other formulae contained in Γ, and Δ.

Intuitively, sequent formulae can be considered as implications. We can read them like "If all of the formulae in the antecedent are true, then at least one of the formulae of the succedent is true". Or written as formula:

$$\varphi_1 \wedge \cdots \wedge \varphi_m \rightarrow \varphi_1 \vee \cdots \vee \varphi_n$$

To satisfy the formula, we either have to find a model that makes at least one formulae of the antecedent *false* or at least one of the formulae of the succedent *true*.

Definition 3.33 (Proof Tree)**.** A *proof tree* is a finite tree (shown with the root at the bottom), such that

- each node of the tree is annotated with a sequent;
- each inner node of the tree is additionally annotated with a name of those calculus rules, defined in Section 3.3.1 and Section 3.3.2, that have at least one premiss. This rule relates the node's sequent to the sequents of its descendants; and
- a leaf node may or may not be annotated with a rule. If it is, it is one of the rules that have no premises, also known as closing rules. The sequent in the leaf then is called an axiom.

A *proof tree for a formula* φ is a proof tree where the root sequent is annotated with $\Rightarrow \varphi$. A *branch* of a proof tree is a path from the root to one of the leaves. A branch is *closed* if the leaf is annotated with one of the closing rules. A proof tree is *closed* if all its branches are closed, i.e., every leaf is annotated with a closing rule. A closed proof tree (for a formula φ) is also called a *proof* (for φ).

[33] This definition is taken from [BHS07], Definition 2.50.

[34] Definition 2.42 of [BHS07] was the guideline for this definition.

3.3.1 Calculus Rules

A calculus provides a set of calculus rules. In this section, we will introduce the most important rules for a dynamic-logic calculus. It is possible to define many more rules, but we just want to give a rough idea of the inner workings of a calculus rather than a detailed specification.

The presented rules are necessary to deal with WHILE programs. The most relevant rules for our approach are the invariant rules for loops which we present in Section 3.3.2.

In the presentation of the calculus rules, some of the occurring formulae are preceded by an update, which is in general denoted by $\mathcal{U}$. The update is used to store intermediate states of the program. In a rule, the set of formulae in the antecedent and succedent which are not touched by the rule are denoted Γ and Δ. These three entities $\mathcal{U}$, Γ and Δ are called the *context* of the sequent. There are rules, which require some of the premises to leave out the context. We will state this case explicitly in the description of an affected rule. An example for such a rule is the invariant rule invRule in Figure 3.8.

First Order Rules In Figure 3.3, we show a set of rules to handle formulae which are built from the classic first-order connectives like $\wedge$ and $\rightarrow$. Formulae in dynamic logic can have preceding updates, for example $\varphi = \mathcal{U}\varphi'$. For all rules we state explicitly which formulae have updates and which not, because otherwise they would be unsound.

There is one rule for the occurrence of a connective for each side of the sequent formula. In Figure 3.4, we state the set of classical first-order closing rules, which can be used to close a proof branch.

Note 3.34 (Substitution Only by Rigid Terms)**.** We demand that free variables are only substituted by rigid terms by the rules, because free variables can also occur in updates or modalities. This restriction is made for convenience reasons, because otherwise certain properties of substitions[35] do not hold anymore and therefore the calculus became more complex.

Note, that the rules allLeft and exRight are slightly enhanced versions of the rules compared to classical first-order calculi[36], because in those rules, skolemization is performed and this process has to collect the free variables that occur in a quantified formula to ensure soundness. These free variables are then the inputs of the newly introduced function symbols. In calculi which only work with closed formulae, this is not necessary.

Equality Rules The idea of the equality rules is, whenever an entity of a formula equals another one, we can substitute one of them for the other in every occurrence. In the presence of a type system, we have to be careful with substitutions of terms. Therefore we do not introduce the common equality rule of traditional text books,

[35] See [Fit96] for an introduction to substitution.

[36] Compared to the calculus in [HKT00], for example.

$$\frac{\Gamma, \varphi, \psi \Rightarrow \Delta}{\Gamma, \varphi \wedge \psi \Rightarrow \Delta}\ \textsf{andLeft} \qquad \frac{\Gamma \Rightarrow \varphi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \wedge \psi, \Delta}\ \textsf{andRight}$$

$$\frac{\Gamma, \varphi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi \vee \psi \Rightarrow \Delta}\ \textsf{orLeft} \qquad \frac{\Gamma \Rightarrow \varphi, \psi, \Delta}{\Gamma \Rightarrow \varphi \vee \psi, \Delta}\ \textsf{orRight}$$

$$\frac{\Gamma \Rightarrow \varphi, \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \varphi \rightarrow \psi \Rightarrow \Delta}\ \textsf{impLeft} \qquad \frac{\Gamma, \varphi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \varphi \rightarrow \psi \Delta}\ \textsf{impRight}$$

$$\frac{\Gamma \Rightarrow \varphi, \Delta}{\Gamma, \neg\varphi \Rightarrow \Delta}\ \textsf{notLeft} \qquad \frac{\Gamma, \varphi \Rightarrow \Delta}{\Gamma \Rightarrow \neg\varphi, \Delta}\ \textsf{notRight}$$

$$\frac{\Gamma, \forall x\ \varphi, [x/t](\varphi) \Rightarrow \Delta}{\Gamma, \forall x\ \varphi \Rightarrow \Delta}\ \textsf{allLeft}$$

with $t \in \mathfrak{T}$ rigid and ground

$$\frac{\Gamma \Rightarrow [x/f(X_1, \ldots, X_n)](\varphi), \Delta}{\Gamma \Rightarrow \forall x\ \varphi, \Delta}\ \textsf{allRight}$$

with $f \in \mathcal{F}$ fresh with output type $\sigma(x)$
and $\{X_1, \ldots, X_n\} = \mathrm{fv}(\varphi)\backslash\{x\}$

$$\frac{\Gamma, [x/f(X_1, \ldots, X_n)](\varphi) \Rightarrow \Delta}{\Gamma, \exists x\ \varphi \Rightarrow \Delta}\ \textsf{exLeft}$$

with $f \in \mathcal{F}$ fresh with output type $\sigma(x)$and $\{X_1, \ldots, X_n\} = \mathrm{fv}(\varphi)\backslash\{x\}$

$$\frac{\Gamma \Rightarrow \exists x\ \varphi, \mathcal{U}[x/t](\varphi), \Delta}{\Gamma \Rightarrow \exists x\ \varphi, \Delta}\ \textsf{exRight}$$

with $t \in \mathfrak{T}$ rigid and ground

Figure 3.3: First-order Rules

$$\frac{}{\Gamma, \mathit{false} \Rightarrow \Delta}\ \textsf{closeFalse} \qquad \frac{}{\Gamma \Rightarrow \mathit{true}, \Delta}\ \textsf{closeTrue} \qquad \frac{}{\Gamma, \varphi \Rightarrow \varphi, \Delta}\ \textsf{close}$$

Figure 3.4: First-order Axioms

$$\frac{\Gamma, t_1 = t_2, [z/t_1](\varphi), [z/t_2](\varphi) \Rightarrow \Delta}{\Gamma, t_1 = t_2, [z/t_1](\varphi) \Rightarrow \Delta}\ \mathsf{eqLeft}$$
$$\text{if } \sigma(t_2) = \sigma(t_1)$$

$$\frac{\Gamma, t_1 = t_2 \Rightarrow [z/t_2](\varphi), [z/t_1](\varphi), \Delta}{\Gamma, t_1 = t_2 \Rightarrow [z/t_1](\varphi), \Delta}\ \mathsf{eqRight}$$
$$\text{if } \sigma(t_2) = \sigma(t_1)$$

$$\frac{\Gamma, t_2 = t_1 \Rightarrow \Delta}{\Gamma, t_1 = t_2 \Rightarrow \Delta}\ \mathsf{eqSymmLeft} \qquad \frac{}{\Gamma \Rightarrow t = t, \Delta}\ \mathsf{eqClose}$$

Figure 3.5: Equality Rules

but add several rules to handle equality while taking care of the types. Figure 3.5 states the different rules for equality.

Rules for Handling Arithmetic Because the logic contains the algebraic functions like for example "+" and "$*$", we need rules to deal with arithmetic. Such rules apply for example the distributive or the commutative law on terms. We do not state any arithmetic rules explicitly here, but assume that they exist in the calculus. The interested reader may have a look at [Rüm07].

Note 3.35 (Division by Zero)**.** Division by zero is handled in the calculus as follows. If a number is divided by zero, the result is `NaN`, an explicit value for "not a number". The application of mathematical operations `NaN` leads to the result of `NaN` as well[37].

$$\texttt{NaN} \circ a = a \circ \texttt{NaN} = \texttt{NaN}$$

for $a \in \{\ldots, \texttt{-1}, \texttt{0}, \texttt{1}, \ldots, \texttt{NaN}\}$ and $\circ \in \{*, /, \%, -\}$. The same holds for the unary minus operation,

$$-\texttt{NaN} = \texttt{NaN}.$$

If `NaN` is compared with $<, \leqslant, \geqslant, >, =, \neq$ to a number, then the result is `false`. For the comparison operation holds

$$\texttt{NaN} \circ \texttt{NaN} = \texttt{true}$$

for $\circ \in \{=, \leqslant, \geqslant\}$ and

$$\texttt{NaN} \circ \texttt{NaN} = \texttt{false}$$

for $\circ \in \{<, \neq, >\}$.

[37] This definition for the handling the division by zero does not comply to how the theorem prover KeY handles it, which we used in our implementation. We chose this definition different from KeY, because otherwise we had to include exceptions into the While language, since Java programs raise an exception when a division by zero occurs. Because division by zero is not a problem in any of our example programs we abstain from quoting a full description and just assume that its definition is sound and consistent.

$$\frac{\Gamma \Rightarrow \mathcal{U}[p]\varphi, \Delta \quad \Gamma \Rightarrow \mathcal{U}\langle p\rangle true, \Delta}{\Gamma \Rightarrow \mathcal{U}\langle p\rangle\varphi, \Delta}\ \mathsf{diamondToBox}$$

$$\frac{\Gamma \Rightarrow \mathcal{U}\varphi, \Delta}{\Gamma \Rightarrow \mathcal{U}\langle\rangle\varphi, \Delta}\ \mathsf{emptyDiamond} \quad \frac{\Gamma \Rightarrow \mathcal{U}\varphi, \Delta}{\Gamma \Rightarrow \mathcal{U}[]\varphi, \Delta}\ \mathsf{emptyBox}$$

Figure 3.6: Rules for Modalities

Modalities The rules emptyDiamond and emptyBox (Figure 3.6) simply remove the empty modalities from the formulae. The rule diamondToBox (Figure 3.6) splits up the proof of total correctness into the branches of the termination and partial correctness[38]. See Note 3.30 for details on total and partial correctness.

Symbolic Execution

Because dynamic-logic formulae can contain programs, the calculus has to be able to deal with them. For this purpose, we introduce a number of rules for symbolic execution. The idea behind those rules is to reduce formulae which contain programs more and more until a first-order formula remains, which then can be handled by the first-order calculus rules. The reduction of programs is done by symbolic execution, which is a technique introduced by [Kin76].

The proof procedure performs symbolic execution of a program by performing the action which is described by the first program statement. Programs in formulae are this way reduced further and further, and the information is saved in state descriptions and case distinctions. State descriptions in our calculus are saved in updates (Definition 3.5) and case distinctions are made by branching.

All rules which deal with programs are applied to the *active statement*, which is the first statement of the program. The rules only perform their action on this statement, usually resulting in a sequent formulae that contains a program which is reduced by this statement. This means that the proof moves forward to the next statement of the program. The rest of the program, which is not the active statement, is denoted by ω in the calculus rules.

When the symbolic execution is performed, more and more information is stored in updates. The calculus transforms every set of updates into a parallel update[39]. These transformations are done by calculus rules as well. For the sake of brevity, we will abstain from stating these rules explicitly here and refer the interested reader to [BHS07], Section 3.6.

Assignments The way we defined programs in Definition 3.10 excludes the appearance of expressions with side-effects, which makes dealing with assignments in the calculus easy. The only necessary rule for dealing with assignments is thus the

[38] This rule is too strict for non-deterministic programs, but this fact is not relevant for our work, because the considered programming languages in this work are all deterministic.

[39] This transformation is always possible, see [Rüm06].

$$\frac{\Gamma \Rightarrow \mathcal{U}\{loc := val\}\langle\omega\rangle\varphi, \Delta}{\Gamma \Rightarrow \mathcal{U}\langle loc = val;\ \omega\rangle\varphi, \Delta}\ \textsf{assignment}$$

$$\frac{\Gamma, \mathcal{U}se = true \Rightarrow \mathcal{U}\langle p\ \omega\rangle\varphi, \Delta \quad \Gamma, \mathcal{U}se = false \Rightarrow \mathcal{U}\langle q\ \omega\rangle\ \varphi, \Delta}{\Gamma \Rightarrow \mathcal{U}\langle \texttt{if (} se \texttt{) } \{\ p\ \} \texttt{ else } \{\ q\ \}\ \omega\rangle\varphi, \Delta}\ \textsf{ifElseSplit}$$

Figure 3.7: Rules for Symbolic Execution

rule assignment in Figure 3.7. It simply turns the assignment into an update of the program variable.

Conditionals Because `if-then-else` statements are included in the While language, we need rules to handle them. The most intuitive one is the rule ifElseSplit in Figure 3.7, which literally splits up the proof tree into a branch where the `if` condition evaluates to *true* and one where it does not.

Note 3.36 (Rules all Types of Modality and Antecedents)**.** For program statements in the rules which we described so far, it does not make a difference if the program is in a box or a diamond modality. It would only matter if there is a chance that the program does not terminate, but this risk only exists for loops. Therefore assume that we have all calculus rules so far for both types of modality, the diamond modality and the box modality.

The same applies for the rules of which we showed only a version where the affected formula is in the succedent of the sequent. Assume that we have a respective rule for the occurrence of the formula in the antecedent as well.

The next section will deal with calculus rules for loops. Assume that the rule loopUnwind exists also for the antecedent and the box modality. The other loop rules, namely all invariant rules occur in the calculus only in the very version which we present here. For those rules it actually matters in which type of modality the program is.

3.3.2 Particular Calculus Rules for Loops

There are two ways to deal with loops in this calculus, unwinding or using an invariant.

Unwinding a Loop We unwind a loop using the rule loopUnwind in Figure 3.8. Of course, this is only possible for a limited number of iterations, thus not applicable for loops whose number of iterations is not specified in the beginning. This rule is often used in combination with induction, but we will not examine this approach any further in this work.

Invariant Rule In case that the loop is in a box modality we can apply the invariant rule invRule. The idea behind this rule is, if we have a formula *inv*, which

$$\frac{\Gamma \Rightarrow \mathcal{U}\langle \mathtt{if}\ \mathtt{(}\ e\ \mathtt{)}\ \{\ p\ \mathtt{while}\ \mathtt{(}\ e\ \mathtt{)}\ \{\ p\ \}\ \}\ \omega\rangle\varphi, \Delta}{\Gamma \Rightarrow \mathcal{U}\langle \mathtt{while}\ \mathtt{(}\ e\ \mathtt{)}\ \{\ p\ \}\ \omega\rangle\varphi, \Delta}\ \mathsf{loopUnwind}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U} inv, \Delta \\ inv, se = true \Rightarrow [p] inv \\ inv, se = false \Rightarrow [\omega]\ \varphi\end{array}}{\Gamma \Rightarrow \mathcal{U}[\mathtt{while}\ \mathtt{(}\ se\ \mathtt{)}\ \{\ p\ \}\ \omega]\varphi, \Delta}\ \mathsf{invRule}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U} inv, \Delta \\ inv, se = true \Rightarrow \langle p\rangle inv \\ inv, se = false \Rightarrow \langle\omega\rangle\varphi \\ c = t, se = true \Rightarrow \langle p\langle t < c\end{array}}{\Gamma \Rightarrow \mathcal{U}\langle \mathtt{while}\ \mathtt{(}\ se\ \mathtt{)}\ \{\ p\ \}\ \omega\rangle\varphi, \Delta}\ \mathsf{invRuleTerm}$$

Figure 3.8: Loop Rules

is an invariant, then we can use it to prove the property behind the modality. To do so, the invariant must be chosen in a way that the proof branches of all of these three premises can be closed:

- *The invariant must be initially valid.* That means, the program state which is reached when we execute the program until right before the `while` statement must fulfill the invariant *inv*. This is expressed in sequent $\Gamma \Rightarrow \mathcal{U} inv, \Delta$. $\mathcal{U}$ is a parallel update describing the state of the program right before the loop and Γ and Δ are the other formulae that have been in the initial sequent. The formula $\mathcal{U} inv$ simply means that *inv* holds after the execution of the program up till the loop. The branch which originates from this premiss is called *init-branch.*
- *The invariant must indeed be an invariant.* That means if the invariant and the loop condition hold, then, after one execution of the loop body (which is done for sure because the loop condition holds), the invariant holds again. This basically says that an arbitrary execution of the loop cannot make the invariant false. Note that the second premiss must have no context, namely formulae like Γ and Δ of the initial sequent nor an update $\mathcal{U}$, because this premiss must hold for any execution of the loop body. If we would include a context, we could not be sure that the information of the context is correct for an arbitrary iteration. The branch which originates from this premiss is called *body-branch.*
- *The invariant and the negated loop condition imply the property φ.* This premiss says that in case the end of the loop is reached (that is when the loop condition becomes false), then the invariant implies the property φ, which is the one we want the program state to fulfill after the execution of the loop. Note, that also here we have to leave out the context $\mathcal{U}$, Δ and Γ of the sequent to make sure that this holds for an arbitrary iteration of the loop. The branch which originates from this premiss is called *use-case-branch.*

```
context(int n) {
  int a = -1;
  while (0 < n) {
    n = n+a;
  }
  return n;
}
```

Figure 3.9: Context
This program's correctness is provable with the rule invRuleMod but not with the rule invRule.

The question is which invariant to use. This is not answered by the calculus and the user has to provide it by herself. For partial correctness proofs, this problem is addressed in this work.

Invariant Rule for Total Correctness In case the program is in a diamond modality, in contrast to the case above, we additionally have to prove that the loop will terminate for sure. In case of the box modality it was only necessary that the property φ is fulfilled *if* the execution terminates. In case it does not terminate, the formulae with the box modality would hold anyway. To prove the termination of a loop we add another premiss to the invariant rule and thus create the rule invRuleTerm in Figure 3.8. The other three premises are the same as for the box modality.

The sequent

$$c = t, se \Rightarrow [p]t < c$$

introduces a term t. This term has to be of a type whose domain is ordered and well-founded. The premiss then states that with each iteration of the loop body the valuation of the term t decreases. Because a well-founded ordering means that no infinite descending chains exist, the term t must finally reach a minimum element. When this element is reached, the term cannot decrease any further and thus the loop terminates. Like the invariant, the termination term t has to be provided by the user.

Improved Invariant Rule with Modifier Set The two invariant rules which we presented so far come with the drawback that in some of the premises the information of the context $\mathcal{U}$, Δ and Γ is dismissed. This is necessary, because the formulae in those branches must hold for any iteration.

Actually, it is not necessary to remove all this information. Consider the example program in Figure 3.9. The input variable `n` is decreased in every iteration until 0 is reached. Assume, that we want to proof that the result of the function is always 0, if `n` was positive in the beginning. The proof obligation would be

$$(\texttt{n} > 0) \rightarrow [\texttt{loop(n);}](n = 0)$$

The construction of the proof makes the application of the invariant rule invRule necessary. The formula $0 \leqslant n$ sounds like a suitable invariant for this problem. After the application of the invariant rule, we yield the following premises.

$$\frac{\begin{array}{c} 0 < \mathtt{n} \Rightarrow \{\mathtt{a} := -1\}\ 0 \leqslant \mathtt{n} \\ 0 \leqslant \mathtt{n},\ 0 < \mathtt{n} = \mathit{true} \Rightarrow [\mathtt{n=n+a}]\ 0 \leqslant \mathtt{n} \\ 0 \leqslant \mathtt{n},\ 0 < \mathtt{n} = \mathit{false} \Rightarrow 0 = \mathtt{n} \end{array}}{0 < \mathtt{n} \Rightarrow \{\mathtt{a} := -1\}[\mathtt{while\ (\ 0 < n\ )\ \{\ n=n+a;\ \}}](\mathtt{n} = 0)}\ \textsf{invRule}$$

Whereas the first and the third premiss can easily be proven, we encounter a problem at the second one. Without the information $\mathtt{a} = -1$, we cannot prove that the addition of `a` to `n` does not violate the invariant $0 \leqslant \mathtt{n}$. This premiss would for example not hold for $\mathtt{n} = 1$ and $\mathtt{a} = -5$. Thus, the information of the value of `a` is essential.

If we include the formula $\mathtt{a} = -1$ in the invariant, the proof can be closed. It is a common problem that such information about variables which are not changed in the loop condition blows up the invariant. The invariant is "polluted" by information what the loop does *not* do in contrast to what it actually does.

If we included the information about the variable `a` in the context in the second premiss, we would be able to solve the problem. We actually *can* do that, because `a` is not manipulated in the loop body, which means the initial value of `a` is the same throughout all iterations. We call the set of variables which is actually changed in the loop body *modifier set*. In the example, the modifier set only contains the variable `n`. The formal definition[40] of modifier sets follows.

Definition 3.37 (Syntax of Modifier Sets)**.** Let $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ be a signature for the type set $\mathcal{T}$. A modifier set $\mathrm{Mod} = \{x_1, \ldots, x_n\} \subseteq \mathcal{V}_p$ is a set of program variables.

Definition 3.38 (Semantics of Modifier Sets)**.** Given a signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$, let $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ be a model, β a logical variables assignment and p a program. A pair of states $(\gamma_1, \gamma_2) = \mathcal{S}^{\mathcal{M}} \times \mathcal{S}^{\mathcal{M}}$ *satisfies* a modifier set Mod

$$(s_1, s_2) \models \mathrm{Mod}$$

iff, for all $v \in \mathcal{V}_p$ the following holds:

$$\gamma_1(v) \neq \gamma_2(v) \Rightarrow v \in \mathrm{Mod}\,.$$

The modifier set Mod is *correct* for the program p, if

$$(\gamma_1, \gamma_2) \models \mathrm{Mod}$$

for all state pairs (γ_1, γ_2), for which

$$\gamma_2 \neq s_\infty$$

and

$$\gamma_2 = [\![p]\!]^{\mathcal{M}}(\gamma_1).$$

[40] The definitions for the syntax and semantics are loosely inspired by Definitions 3.61 and 3.62 of [BHS07].

A minimal modifier set of a program contains only the variables which are actually changed in the program and nothing more.

Definition 3.39 (Minimal Modifier Set)**.** Let p be a program. A *minimal modifier set* Mod_p of a program p does not contain any variables $v \in \mathcal{V}_p$ for which:

$$\gamma_1(v) = \gamma_2(v)$$

for all pairs of states $(\gamma_1, \gamma_2) \in \mathcal{S} \times \mathcal{S}$, where γ_1 is a start state of the program and γ_2 is the result state of the program.

The idea behind the improved invariant rule is to maintain the context of the sequent as much as possible. That means we keep all variable assignments, but erase the contents of variables in the modifier set. This erasing is done via anonymizing updates. Anonymizing updates have the same form as normal updates (see Definition 3.5), except that they assign a value which is fixed, but unknown. The formal definition[41] for anonymizing updates with respect to modifier sets follows.

Definition 3.40 (Anonymizing Update)**.** Let $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ be a signature for the type set $\mathcal{T}$, a WHILE program p, a sequent $\Gamma \Rightarrow \varphi, \Delta$, an invariant φ (and for the rule invRuleTerm also a term t) be given. For every $v \in \mathcal{V}_p$ occurring in p or in $\Gamma \cup \Delta \cup \{\varphi\}$ let $v^* \in \mathcal{F}$ be a fresh function symbol (fresh with respect to p and $\Gamma \cup \Delta \cup \{\varphi\}$) of the same type as v. Then the update

$$v := v^*(X_1, \ldots, X_n)$$

is called *anonymizing update* for the sequent $\Gamma \Rightarrow \varphi, \Delta$. The inputs $X_1, \ldots, X_n$ are the free variables of the formulae and updates inv,$\mathcal{U}$ and φ (and t)[42]. In the following we abbreviate an anonymizing update with $\mathcal{U}^*$.

Definition 3.41 (Anonymizing Update w.r.t. a Modifier Set)**.** Let $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ be a signature for the type set $\mathcal{T}$, a modifier set Mod, a sequent $\Gamma \Rightarrow \varphi, \Delta$, an invariant φ (and for the rule invRuleTerm also a term t) be given. Then the anonymizing update with respect to the modifier set Mod, $\mathcal{U}^*(\mathrm{Mod})$ is a parallel update

$$u_1 \mid\mid \ldots \mid\mid u_n$$

with u_v being an anonymizing update for each of the variable $v \in \mathrm{Mod}$.

With this notion of anonymizing updates, we can form a new invariant rule invRuleMod. This rule keeps the context of the initial sequent, but introduces an anonymizing update with respect to the minimal modifier set of the program p in the loop body of the loop. The rule is shown in Figure 3.10.

This improvement can also be done to the invariant rule for the diamond modality invRuleTerm. The modified rule is the rule invRuleTermMod in Figure 3.10. With this modified invariant rule, it is no problem to prove the correctness of the example program in Figure 3.9.

[41] This definition is taken from [BHS07], Definition 3.64.

[42] This collecting of variables is necessary when working with metavariables to ensure the soundness of the calculus rules.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}\mathit{inv}, \Delta \\ \Gamma, \mathcal{A}\mathit{inv}, \mathcal{A}\mathit{se} = \mathit{true} \Rightarrow \mathcal{A}[p]\ \mathit{inv}, \Delta \\ \Gamma, \mathcal{A}\mathit{inv}, \mathcal{A}\mathit{se} = \mathit{true} \Rightarrow \mathcal{A}\varphi, \Delta\end{array}}{\Gamma \Rightarrow \mathcal{U}[\texttt{while (} \mathit{se} \texttt{) } \{\ p\ \}]\ \varphi, \Delta}\ \textsf{invRuleMod}$$

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}\mathit{inv}, \Delta \\ \Gamma, \mathcal{A}\mathit{inv}, \mathcal{A}\mathit{se} = \mathit{true} \Rightarrow \mathcal{A}\langle p\rangle\ \mathit{inv}, \Delta \\ \Gamma, \mathcal{A}\mathit{inv}, \mathcal{A}\mathit{se} = \mathit{false} \Rightarrow \mathcal{A}\varphi \\ \Gamma, \mathcal{A}c = t, \mathcal{A}\mathit{se} = \mathit{true} \Rightarrow \mathcal{A}\langle p\rangle\ t < c, \Delta\end{array}}{\Gamma \Rightarrow \mathcal{U}\langle\texttt{while (} \mathit{se} \texttt{) } \{\ p\ \}\rangle\ \varphi, \Delta}\ \textsf{invRuleTermMod}$$

where $\mathcal{A} = \mathcal{U}\mathcal{U}^*(\mathrm{Mod})$ and Mod is a correct modifier set for p.

Figure 3.10: Invariant Rules with Anonymous Updates

```
gaussCorrect(int n) {
  int i = 0;
  int res = 0;
  if (n > 0) {
    while (i < n) {
      res += i;
      i++;
    }
  }
  return res;
}
```

Figure 3.11: GaussCorrect

This program calculates the Gaussian sum, namely the sum of all integers between 0 and the value of the input variable `n`. The program does terminate for all inputs.

3.3.3 Example Proof

Now that we have all necessary calculus rules, we show a nearly complete example proof. We say "nearly", because we will leave out some steps which represent some obvious algebraic transformations. The program which we want to reason about is the program GaussCorrect which is shown in Figure 3.3.3. The program sums up all numbers between zero and the value of the input variable `n`. This is called the Gaussian sum and Carl Friedrich Gauss is known to have found the following concise expression for this sum as a 12-year-old.

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{3.1}$$

We prove that the calculation of the Gaussian sum in our example program is according to this formula. Thus, we form the proof obligation

$$\texttt{n} > 0 \rightarrow \langle \texttt{r = gaussCorrect(n);} \rangle\ \texttt{r} = \texttt{n} * (\texttt{n} - 1)/2. \tag{3.2}$$

The construction of the proof begins with symbolic execution of the program. Because the first two statements of the program are assignments, we apply the rule assignment twice and yield a parallel update for the variables `i` and `res`. The proof so far looks like this (to be read from bottom to top).

$$\begin{array}{c}
\mathtt{n} > 0 \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\langle\texttt{if (n > 0) \{ }\ldots\texttt{ \}}\rangle\, r = n * (n-1)/2 \\
\hline
\mathtt{n} > 0 \Rightarrow \{\mathtt{i} := 0\}\langle\texttt{int res = 0;}\ldots\rangle\, r = n * (n-1)/2 \\
\hline
\mathtt{n} > 0 \Rightarrow \langle\texttt{int i = 0;}\ldots\rangle\, r = n * (n-1)/2 \\
\hline
\mathtt{n} > 0 \Rightarrow \langle\texttt{r = gauss(n);}\rangle\, r = n * (n-1)/2
\end{array} \tag{3.3}$$

At this point in the proof construction, the active statement of the program is the `if`-statement. We apply the ifElseSplit rule and the proof branches into these two branches:

$$\begin{array}{c}
\mathtt{n} > 0, \mathtt{n} > 0 \Rightarrow \\
\{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\langle\texttt{while (i < n) \{ }\ldots\texttt{ \}}\rangle\, r = n * (n-1)/2
\end{array} \tag{3.4}$$

and

$$\mathtt{n} > 0, \mathtt{n} \leqslant 0 \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\langle\rangle\, r = n * (n-1)/2. \tag{3.5}$$

We will first explore the branch of sequent (3.5), because it is the shorter one.

After the application of the ifElseSplit rule, the program is reduced to an empty modality $\langle\rangle$, which we can remove via the rule emptyDiamond.

The sequent contains the two formulae $\mathtt{n} > 0$ and $\mathtt{n} \leqslant 0$. The first one is equivalent to $1 \leqslant \mathtt{n}$. This formula together with $\mathtt{n} \leqslant 0$ can be used to derive the formula $1 \leqslant \mathtt{n} \leqslant 0$, thus $1 \leqslant 0$, which is of course equivalent to *false*. Once, there is the formula *false* in the antecedent, we can apply the closing rule closeFalse to close the branch. Without the intermediate steps for the handling of the arithmetic, the proof branch looks like this (to be read from bottom to top).

$$\begin{array}{c}
\hline
1 \leqslant \mathtt{n}, \mathtt{n} \leqslant 0, \mathit{false} \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\; r = n * (n-1)/2 \\
\hline
1 \leqslant \mathtt{n}, \mathtt{n} \leqslant 0, 1 \leqslant 0 \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\; r = n * (n-1)/2 \\
\hline
1 \leqslant \mathtt{n}, \mathtt{n} \leqslant 0 \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\; r = n * (n-1)/2 \\
\hline
\mathtt{n} > 0, \mathtt{n} \leqslant 0 \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\langle\rangle\, r = n * (n-1)/2
\end{array} \tag{3.6}$$

Now we go back to sequent (3.4), which was the sequent

$$\mathtt{n} > 0 \Rightarrow \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\}\langle\texttt{while (i < n) \{ }\ldots\texttt{ \}}\rangle\, r = n * (n-1)/2.$$

The `while` statement is the active statement, which means we have to apply the rule invRuleTermMod. We choose the second one to show the application of the invariant rule with a modifier set. The context of the sequent is here:

$$\begin{array}{c}
\Gamma = \{\mathtt{n} > 0\} \\
\mathcal{U} = \{\mathtt{i} := 0 \parallel \mathtt{res} := 0\} \\
\Delta = \{\}
\end{array} \tag{3.7}$$

The minimum modifier set of the loop body is Mod $= \{\mathtt{res}, \mathtt{i}\}$. The adequate anonymizing update is

$$\mathcal{A} = \mathcal{U}^*(\mathrm{Mod}) = \{\mathtt{i} := i^* \mid\mid \mathtt{res} := res^*\}, \tag{3.8}$$

where i^* and res^* are freshly introduced function symbols.

For the application of the rule, we need an invariant inv and a ranking term t. A suitable invariant is

$$inv \equiv \mathtt{res} = \mathtt{i} * (\mathtt{i} - 1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n} \tag{3.9}$$

and an adequate ranking term is

$$t = \mathtt{n} - \mathtt{i}. \tag{3.10}$$

The reader might wonder why those two things are suitable for the proof. The answer is, because they actually work. Finding the invariant and the ranking term are non-trivial tasks. If there is no procedure which provides those, the user has to find them by herself. Usually this means constructing it by intuition or from experience. This thesis is about invariant generation, so actually part of this problem is solved for at least non-termination proofs. But in this case, we go on with the invariant and ranking term that magically appeared in the proof.

After the application of the invariant rule, we obtain the four sequents:

$$\mathtt{n} > 0 \Rightarrow \{\mathtt{i} := 0 \mid\mid \mathtt{res} := 0\}\ inv \tag{3.11}$$

$$\mathtt{n} > 0, \mathcal{A}inv, \mathcal{A}\mathtt{i} < \mathtt{n} \Rightarrow \mathcal{A}\langle \mathtt{res\ +=\ i;} \ldots \rangle\ inv \tag{3.12}$$

$$\mathtt{n} > 0, \mathcal{A}inv, \mathcal{A}\mathtt{i} \geqslant \mathtt{n} \Rightarrow \mathcal{A}\mathtt{res} = \mathtt{n} * (\mathtt{n} - 1)/2 \tag{3.13}$$

$$\mathtt{n} > 0, \mathcal{A}c = \mathtt{n} - \mathtt{i}, \mathcal{A}\mathtt{i} < \mathtt{n} \Rightarrow \mathcal{A}\langle \mathtt{res\ +=\ i;} \ldots \rangle\ \mathtt{n} - \mathtt{i} < c \tag{3.14}$$

We slightly cheated here, because we did not introduce the result variable `r` of the proof obligation, but assumed that we want to proof the post condition $\mathtt{res} = \mathtt{n} * (\mathtt{n} - 1)/2$. These branches look pretty ugly so far, but we will reduce the complexity by dealing with each branch separately[43].

The sequent (3.11) says that the invariant has to be initially valid.

$$\mathtt{n} > 0 \Rightarrow \{\mathtt{i} := 0 \mid\mid \mathtt{res} := 0\}\ \mathtt{res} = \mathtt{i} * (\mathtt{i} - 1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n}$$

We apply the update $\{\mathtt{i} := 0 \mid\mid \mathtt{res} := 0\}$ and obtain the following sequent. We have not stated rules for the update application explicitly, but intuitively it is the transference of the information in the updates into the formula.

$$\mathtt{n} > 0 \Rightarrow 0 = 0 * (0 - 1)/2 \wedge 0 \leqslant 0 \wedge 0 \leqslant \mathtt{n}$$

[43] When applying an automatic theorem prover, it is not reasonable to first finish one branch before examining the next. Usually, all branches are expanded by iteratively applying one rule to the first, one to the second and so on. The reason is that some extensions to the proof procedure, for example the constraint solver, need to have some fairness conditions for all branches. For details see Chapter 3.4. Because those facts are not relevant here, we examine each branch independently of the others.

Unfortunately, a conjunction in the succedent of a sequent requires the proof to branch. The repeated application of the rule andRight yields the following branches.

$$\texttt{n} > 0 \Rightarrow 0 = 0 * (0 - 1)/2 \tag{3.15}$$

$$\texttt{n} > 0 \Rightarrow 0 \leqslant 0 \tag{3.16}$$

$$\texttt{n} > 0 \Rightarrow 0 \leqslant \texttt{n} \tag{3.17}$$

The formula $0 = 0 * (0 - 1)/2$ in the succedent of sequent (3.15) can easily be shown to be true by the rules for arithmetic. Thus, we obtain the formula *true* in the succedent and can close this branch by the rule closeTrue.

Sequent (3.16) was the following.

$$\texttt{n} > 0 \Rightarrow 0 \leqslant 0$$

Again, the formula of the succedent can easily be shown to be true and thus the branch can be closed by the rule closeTrue.

The third sequent, the sequent (3.17) was this one.

$$0 < \texttt{n} \Rightarrow 0 \leqslant \texttt{n}$$

The formula $0 \leqslant \texttt{n}$ can be split up into the the disjunction $0 < \texttt{n} \vee 0 = \texttt{n}$. The disjunction in the succedent can simply be written as two separate formulae. Then the formula $0 < \texttt{n}$ occurs on both sides of the sequent, which means that we can close the proof by the application of the closing rule close. These steps look in the proof tree like this (read from bottom to top).

$$\cfrac{\cfrac{\overline{0 < \texttt{n} \Rightarrow 0 < \texttt{n}, 0 = \texttt{n}}}{0 < \texttt{n} \Rightarrow 0 < \texttt{n} \vee 0 = \texttt{n}}}{0 < \texttt{n} \Rightarrow 0 \leqslant \texttt{n}} \tag{3.18}$$

With these three branches we have closed the first branch of the application of the invariant rule.

The second branch of the proof after the application invariant rule, sequent (3.12), was this one

$$n > 0, \mathcal{A}inv, \mathcal{A}\texttt{i} < \texttt{n} \Rightarrow \mathcal{A}\langle\texttt{res += i;}\ldots\rangle\ inv$$

which is fully written out:

$$\begin{gathered}\texttt{n} > 0,\\ \{\texttt{i} := i^* \mid\mid \texttt{res} := res^*\}\texttt{res} = \texttt{i} * (\texttt{i} - 1)/2 \wedge 0 \leqslant \texttt{i} \wedge \texttt{i} \leqslant \texttt{n},\\ \{\texttt{i} := i^* \mid\mid \texttt{res} := res^*\}\texttt{i} < \texttt{n}\\ \Rightarrow\\ \{\texttt{i} := i^* \mid\mid \texttt{res} := res^*\}\langle\texttt{res += i;}\ldots\rangle\ \texttt{res} = \texttt{i} * (\texttt{i} - 1)/2 \wedge 0 \leqslant \texttt{i} \wedge \texttt{i} \leqslant \texttt{n}.\end{gathered}$$

In this branch, the symbolic execution of the program is not finished yet. Therefore we apply the rule assignment two more times and remove the resulting empty diamond with the rule emptyDiamond. The next step in the antecedent of the sequent is the application of the update $\mathtt{i} := i^* \mathbin{||} \mathtt{res} := \mathit{res}^*$. The repeated application of the rule andLeft makes the parts of the conjunction in the antecedent appear as single formulae. Eventually, we will apply the update $\{\mathtt{i} := i^*+1 \mathbin{||} \mathtt{res} := \mathit{res}^*+i^*\}$ on the formula in the succedent.

$$\begin{array}{c}
\ldots \Rightarrow \mathit{res}^* + i^* = (i^*+1) \ast (i^*+1-1)/2 \wedge 0 \leqslant i^* \wedge i^* \leqslant \mathtt{n} \\
\hline
\ldots \Rightarrow \{\mathtt{i} := i^*+1 \mathbin{||} \mathtt{res} := \mathit{res}^*+i^*\}\ \mathtt{res} = \mathtt{i} \ast (\mathtt{i}-1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n} \\
\hline
\ldots \Rightarrow \{\mathtt{i} := i^*+1 \mathbin{||} \mathtt{res} := \mathit{res}^*+i^*\}\langle\rangle\ \mathtt{res} = \mathtt{i} \ast (\mathtt{i}-1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n} \\
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2, 0 \leqslant i^*, i^* \leqslant \mathtt{n} \\
\Rightarrow \{\mathtt{i} := i^* \mathbin{||} \mathtt{res} := \mathit{res}^*+i^*\}\langle\mathtt{i++;}\rangle\ \mathtt{res} = \mathtt{i} \ast (\mathtt{i}-1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n} \\
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2 \wedge 0 \leqslant i^* \wedge i^* \leqslant \mathtt{n} \\
\Rightarrow \{\mathtt{i} := i^* \mathbin{||} \mathtt{res} := \mathit{res}^*\}\langle\mathtt{res\ +=\ i;}\ldots\rangle\ \mathtt{res} = \mathtt{i} \ast (\mathtt{i}-1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n} \\
\hline
\mathtt{n} > 0, \{\mathtt{i} := i^* \mathbin{||} \mathtt{res} := \mathit{res}^*\}\mathtt{res} = \mathtt{i} \ast (\mathtt{i}-1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n} \\
\Rightarrow \{\mathtt{i} := i^* \mathbin{||} \mathtt{res} := \mathit{res}^*\}\langle\mathtt{res\ +=\ i;}\ldots\rangle\ \mathtt{res} = \mathtt{i} \ast (\mathtt{i}-1)/2 \wedge 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant \mathtt{n}
\end{array} \tag{3.19}$$

The resulting sequent contains again a conjunction in the succedent, which makes the proof branch by the application of the rule andRight. We obtain three branches.

$$\ldots \Rightarrow \mathit{res}^* + i^* = (i^*+1) \ast (i^*+1-1)/2 \tag{3.20}$$

$$\ldots \Rightarrow 0 \leqslant i^* \tag{3.21}$$

$$\ldots \Rightarrow i^* \leqslant \mathtt{n} \tag{3.22}$$

The sequent (3.20) leads to this sequent.

$$\begin{array}{c}
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2 \wedge 0 \leqslant i^* \wedge i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow \\
\mathit{res}^* + i^* = (i^*+1) \ast (i^*+1-1)/2
\end{array} \tag{3.23}$$

The formula $\mathit{res}^* + i^* = (i^*+1) \ast (i^*+1-1)/2$ in the succedent can be transformed by rules for arithmetic into the formula $\mathit{res}^* = i^* \ast (i^*-1)/2$. The latter also occurs in the antecedent and thus we can close this branch by application of the rule close.

From the sequent (3.21) we obtain this sequent.

$$\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2 \wedge 0 \leqslant i^* \wedge i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow 0 \leqslant i^* + 1 \tag{3.24}$$

We use rules for arithmetic to close this branch in the following manner. (Read from bottom to top.)

$$\begin{array}{c}
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2, i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow -1 \leqslant i^*, -1 \geqslant i^*, \mathit{true} \\
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2, i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow -1 \leqslant i^*, -1 \geqslant i^*, -1 \leqslant -1 \\
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2, i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow -1 \leqslant i^*, -1 \geqslant i^* \\
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2, i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow 0 \leqslant i^* + 1, 0 > i^* \\
\hline
\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2, 0 \leqslant i^*, i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \Rightarrow 0 \leqslant i^* + 1
\end{array}$$

The sequent (3.22) yields this sequent.

$$\mathtt{n} > 0, \mathit{res}^* = i^* \ast (i^*-1)/2 \wedge 0 \leqslant i^* \wedge i^* \leqslant \mathtt{n}, i^* < \mathtt{n} \ldots \Rightarrow i^* + 1 \leqslant \mathtt{n} \tag{3.25}$$

Using the rule andLeft and rules for arithmetic, we can close this branch in a similar manner as the one of sequent (3.21) using the formulae $i^* \leqslant \texttt{n}$ and $i^* + 1 \leqslant \texttt{n}$.

The third branch of the proof after the application of the invariant rule, sequent (3.13) yielded the sequent, which states that with the negation of the loop condition the postcondition holds. Written out the sequent is this one.

$$\begin{array}{c} \texttt{n} > 0, \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}\ \texttt{res} = \texttt{i} * (\texttt{i} - 1)/2 \wedge 0 \leqslant \texttt{i} \wedge \texttt{i} \leqslant \texttt{n}\ \ , \\ \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}\ \texttt{i} \geqslant \texttt{n} \\ \Rightarrow \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}\ \texttt{res} = \texttt{n} * (\texttt{n} - 1)/2 \end{array}$$

After the application of the updates and the rule andLeft we obtain the following sequent.

$$\texttt{n} > 0, \mathit{res}^* = i^* * (i^* - 1)/2, 0 \leqslant i^*, i^* \leqslant \texttt{n}, i^* \geqslant \texttt{n} \Rightarrow \mathit{res}^* = \texttt{n} * (\texttt{n} - 1)/2$$

From the two formulae $i^* \leqslant \texttt{n}$ and $i^* \geqslant \texttt{n}$, rules for arithmetic can derive the formula $i^* = \texttt{n}$. This equality is used in the application of the rule eqRight and yields the formula $\mathit{res}^* = i^* * (i^* - 1)/2$ in the succedent. This formulae occurs in the antecedent as well and thus we can close the proof by application of close. Written in the proof, these operations are represented in this branch (read from bottom to top).

$$\begin{array}{c} \hline \texttt{n} > 0, \mathit{res}^* = i^* * (i^* - 1)/2, 0 \leqslant i^*, i^* \leqslant \texttt{n}, i^* \geqslant \texttt{n}, i^* = \texttt{n} \Rightarrow \mathit{res}^* = i^* * (i^* - 1)/2 \\ \hline \texttt{n} > 0, \mathit{res}^* = i^* * (i^* - 1)/2, 0 \leqslant i^*, i^* \leqslant \texttt{n}, i^* \geqslant \texttt{n}, i^* = \texttt{n} \Rightarrow \mathit{res}^* = \texttt{n} * (\texttt{n} - 1)/2 \\ \hline \texttt{n} > 0, \mathit{res}^* = i^* * (i^* - 1)/2, 0 \leqslant i^*, i^* \leqslant \texttt{n}, i^* \geqslant \texttt{n} \Rightarrow \mathit{res}^* = \texttt{n} * (\texttt{n} - 1)/2 \end{array} \tag{3.26}$$

The fourth branch of the proof after the application of the invariant rule starts with sequent (3.14), which is this one.

$$\begin{array}{c} \texttt{n} > 0, \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}c = \texttt{n} - \texttt{i}, \\ \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}\texttt{i} < \texttt{n} \\ \Rightarrow \\ \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}\langle \texttt{res += i;} \ldots \rangle\ \texttt{n} - \texttt{i} < c \end{array}$$

After the application of the updates to the formulae of the antecedent we obtain

$$\texttt{n} > 0, c = \texttt{n} - i^*, i^* < \texttt{n} \Rightarrow \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^*\}\langle \texttt{res += i;} \ldots \rangle\ \texttt{n} - \texttt{i} < c$$

We first continue with the symbolic execution of the program, which changes the update of the variables `res` and `i`. We remove the empty diamond and apply the update to the formula $\texttt{n} - i^* < c$, which yields the formula $\texttt{n} - \texttt{i} < c + 1$ after some transformations. Then, rules for arithmetic can easily close the branch. Because of space limitations we abstain from stating all steps explicitly here. (Read from bottom to top.)

$$\begin{array}{c} \ldots \\ \hline \texttt{n} > 0, c = \texttt{n} - i^*, i^* < \texttt{n} \Rightarrow \texttt{n} - i^* < c + 1 \\ \hline \texttt{n} > 0, c = \texttt{n} - i^*, i^* < \texttt{n} \Rightarrow \texttt{n} - (i^* + 1) < c \\ \hline \texttt{n} > 0, c = \texttt{n} - i^*, i^* < \texttt{n} \Rightarrow \{\texttt{i} := i^* + 1 \mid\mid \texttt{res} := \mathit{res}^* + i^*\}\texttt{n} - \texttt{i} < c \\ \hline \texttt{n} > 0, c = \texttt{n} - i^*, i^* < \texttt{n} \Rightarrow \{\texttt{i} := i^* + 1 \mid\mid \texttt{res} := \mathit{res}^* + i^*\}\langle\rangle\ \texttt{n} - \texttt{i} < c \\ \hline \texttt{n} > 0, c = \texttt{n} - i^*, i^* < \texttt{n} \Rightarrow \{\texttt{i} := i^* \mid\mid \texttt{res} := \mathit{res}^* + i^*\}\langle \texttt{i++;} \rangle\ \texttt{n} - \texttt{i} < c \end{array} \tag{3.27}$$

With this branch we have closed the last open branch and thus closed the whole proof. Unfortunately, we cannot display the proof as a tree here because of space limitations.

3.3.4 Properties of the Calculus

Two important properties of proof systems like a calculus are *soundness* and *completeness*. Soundness of a calculus means: If we can prove the validity of a formula using the calculus, then the formula is actually valid. Completeness means: For every valid formula we can prove its validity using the calculus. Soundness is the more important property, because there is usually less harm in not being able to prove something than in proving something that is not actually valid.

The presented calculus is sound. The soundness of the calculus can be proven by proving the soundness of each single rule of the calculus. To do so is beyond the scope of this thesis and the interested reader is referred to for example [BHS07], various papers of the KEY PROJECT[44] and books concerning traditional dynamic-logic, for instance [HKT00].

The calculus can of course not be complete since it deals with arithmetic. Arithmetic is inherently incomplete, which Kurt Gödel proved in his incompleteness theorem[45]. Besides the arithmetic, there is the halting problem which would also be solved, if the calculus was complete. Thus a calculus like ours can never be complete.

Despite this inherent incompleteness, Beckert et al. define a *relative completeness*, which means that the calculus can prove any formula whose proof does not require the derivation of a non-provable first-order property ([BHS07], Proposition 3.47).

3.4 Incremental Closure of Proofs

A first-order calculus (and thus also a dynamic-logic calculus) has to handle formulae with existential quantifiers. This is not a trivial task, because it corresponds to a search for an adequate term in a possibly large search space. Because quantified formulae are an essential part of non-termination proofs, we describe how the proof procedure handles them.

3.4.1 Existentially Quantified Formulae and Metavariables

Have a look at the following example of an existentially quantified formula, which states a property in the domain of integers.

$$\exists x \; \exists y \; (x - y = 10 \land 3 * y = x)$$

To prove the validity of this formula, we have to find at least one integer valued term for each of the variables x and y, for which the subformula $x - y = 10 \land 3 * y = x$

[44] `http://www.key-project.org`

[45] See [Göd31].

$$\frac{\Gamma \Rightarrow \exists x\ \varphi, [x/M](\varphi), \Delta}{\Gamma \Rightarrow \exists x\ \varphi, \Delta}\ \mathsf{exRightMeta}$$

$$\frac{\Gamma, \forall x\ \varphi, [x/M](\varphi) \Rightarrow \Delta}{\Gamma, \forall x\ \varphi \Rightarrow \Delta}\ \mathsf{allLeftMeta}$$

where $M \in \mathcal{V}_m$ is a fresh metavariable.

Figure 3.12: Calculus Rules for the Introduction of Metavariables

holds. Thus, proving the formula is equivalent to a search in the set $\mathcal{D}_{\texttt{int}} \times \mathcal{D}_{\texttt{int}}$. If we find these terms and instantiate the quantified variables with them, we have proven their existence and closed the proof.

Because the decision with which term to instantiate the quantified variable is such a tricky one, most calculi try to postpone it as far as possible to first gain more information from the rest of the proof. A standard way to postpone the instantiation is to introduce metavariables[46].

The insertion of metavariables replaces the existentially quantified variable and removes the quantifier. Metavariables are another kind of logical variables. They are rigid and they stand for a fixed, but not yet specified value. In our calculus, metavariables are treated in the calculus rules the same way as other logical variables. They are the only free variables that can occur proofs. The formal definition of metavariables follows.

Definition 3.42 (Metavariables)**.** Given a signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$, the set of metavariables $\mathcal{V}_m$ is a distinguished subset of $\mathcal{V}_l$.

$$\mathcal{V}_m \subseteq \mathcal{V}_l$$

We extend the calculus which we described in Section 3.3 by the rules exRightMeta and allLeftMeta shown in Figure 3.12. These rules insert the metavariables into formula with existentially quantified variables.

After the instantiation of x with the metavariable X and y with Y, the example formula looks like the following:

$$X - Y = 10 \wedge 3 * Y = X$$

Metavariables have the advantage that the prover no longer has to handle the bulky quantifiers, and thus can apply much more rules than before. However, postponing the decision does not make it disappear. Usually, metavariables are carried along in the proof construction until the proof procedure either terminates with an open goal or can close the proof by different means than the instantiation of the metavariable.

[46] See [Fit96] for an introduction about metavariables.

In the example, the prover would end up with these open goals.

$$X - 10 = Y \text{ and } 3 * Y = X$$

At this point in the proof, the prover has to perform an instantiation as the only chance to finish the proof. One strategy is, to substitute the metavariable by some term and try to close the proof. In case closure is not possible, the prover has to backtrack and try another term. When using backtracking there must be a limit in the number of tries, otherwise it is possible that the proof procedure itself does not terminate. Another drawback of backtracking is that a lot of calculations might be done over an over again, because the information of tracks that lead to failure of a proof attempt is not saved and thus not reused.

3.4.2 Incremental Closure of Proofs

Martin Giese described in [Gie01] an alternative approach to this problem, which removes the need for backtracking. The idea is, to extract requirements for the substitution of the metavariables from the formulae in open goals and store them as constraints. The proof procedure closes the proof when compatible and satisfiable constraints are found which close all branches of the proof at the same time.

Let us illustrate the method with the following example[47], where c and d are constants of type integer and f is a unary function symbol with input type integer.

$$\Rightarrow \exists x\ ((x = c \lor x = d) \land f(c) = f(x))$$

A prover which uses our calculus would construct the following proof.

$$\cfrac{\cfrac{\cfrac{\Rightarrow X = c, X = d}{\Rightarrow X = c \lor X = d} \quad \Rightarrow f(c) = f(X)}{\Rightarrow (X = c \lor X = d) \land f(c) = f(X)}}{\Rightarrow \exists x\ ((x = c \lor x = d) \land f(c) = f(x))}$$

The proof is constructed as far as possible and we can easily read the requirements for the substitution of the metavariable X to close the proof from the open goals. Obviously, the substitution $\{X \mapsto c\}$ would be the reasonable choice. Following Giese's approach, we create unification constraints which describe conditions for X and attach them to the proof branches. The addition of constraints to the proof branches is done via calculus rules, which are for example stated in [Rüm08], Section 3.4.

In our example, the proof tree which is enriched by the constraints looks like this.

$$\cfrac{\cfrac{\cfrac{\cfrac{[X = c], [X = d]}{\Rightarrow X = c, X = d}}{\Rightarrow X = c \lor X = d} \quad \cfrac{[f(c) = f(X)]}{\Rightarrow f(c) = f(X)}}{\Rightarrow (X = c \lor X = d) \land f(c) = f(X)}}{\Rightarrow \exists x\ ((x = c \lor x = d) \land f(c) = f(x))}$$

[47] This example is taken from [Rüm08].

The constraints of one branch must be considered as disjunctions, so either $X = c$ or $X = d$ (or both) have to be solved. The constraints of different branches are considered conjunctively, which means in a search for a solution of the constraints a solution must be found which solves at least one constraint of each branch. Our example yields the overall constraint:

$$(X = c \lor X = d) \land f(c) = f(X)$$

After the creation of constraints in the proof construction, we have to check whether the constraint is solvable. A solution of constraints is a substitution of the metavariables by terms. Fortunately, it is sufficient to know that a solution exists, rather than finding a particular one, because the metavariables came from the existentially quantified formulae, which say "*There is* an x for which ...".

The idea is to invoke a constraint solver after each creation of constraints (or after closure of a branch by other means) to check if the constraints are already solvable. As soon as the constraint solver announces that there is a solution for the constraints which closes all branches of the proof, the proof procedure closes the proof with the set of constraints which were solvable. We explain in the succeeding section, when a set of constraints is solvable.

Because of this possibility to close a proof (branch), we define another notion of closed proof, namely closed by constraints.

Definition 3.43 (Closed by Constraints)**.** A proof (or proof branch or proof subtree) is called *closed by constraints*, if there is a solution for the constraints of all open branches, which closes all open branches at the same time.

Example 3.44 (Closed by Constraints)**.** Have a look at this formula:

$$\exists x\ (x > 5 \land x < 23) \land x = 42, \tag{3.28}$$

where x is of type `int`. After the introduction of the metavariable X for x with the rule exRightMeta, the formula look like this:

$$(X > 5 \land X < 23) \land X = 42.$$

We apply the rule andRight two times and yield a proof tree with three branches with the annotated constraints for the metavariable X.

$$\dfrac{\dfrac{\dfrac{\dfrac{[X > 5]}{\Rightarrow X > 5} \quad \dfrac{[X < 23]}{\Rightarrow X < 23}}{\Rightarrow X > 5 \land X < 23} \quad \dfrac{[X = 42]}{\Rightarrow X = 42}}{\Rightarrow (X > 5 \land X < 23) \land X = 42}}{\Rightarrow \exists x\ (x > 5 \land x < 23) \land x = 42}$$

The constraints of the proof subtree starting with the sequent $\Rightarrow X > 5 \land X < 23$ is $X > 5$ and $X < 23$. These constraints are solvable, because for example $X = 10$ is a solution of them. Thus, this proof subtree is closed by constraints.

The whole proof tree, starting with sequent (3.28) has the constraints $X > 5$, $X < 23$ and $X = 42$ for the metavariable X. For these constraints, there is no solution, because no integer can be equal to 42 and smaller than 23. Thus, this proof tree as a whole is not closed and especially not closed by constraints.

$$\begin{array}{lcl}
\mathsf{UnificationConstraint} & ::= & \mathsf{unification} \mid \\
 & & \mathsf{UnificationConstraint} \wedge \mathsf{UnificationConstraint} \\
\mathsf{unification} & ::= & \mathsf{term} \equiv \mathsf{term}
\end{array}$$

Figure 3.13: Unification Constraints

$$\begin{array}{rcl}
\mathsf{LinearConstraint} & ::= & \mathsf{intConstraint} \\
 & & \mid \mathsf{othConstraint} \\
 & & \mid \mathsf{LinearConstraint} \wedge \mathsf{LinearConstraint} \\
 & & \\
\mathsf{intConstraint} & ::= & \mathsf{intTerm} = \mathsf{intTerm} \\
 & & \mid \mathsf{intTerm} \neq \mathsf{intTerm} \\
 & & \mid \mathsf{intTerm} < \mathsf{intTerm} \\
 & & \mid \mathsf{intTerm} \leqslant \mathsf{intTerm} \\
 & & \\
\mathsf{othConstraint} & ::= & \mathsf{othTerm} = \mathsf{othTerm}
\end{array}$$

Figure 3.14: Linear Unification Constraints

Constraint Types

A question is, what kind of constraints to extract from formulae in open goals. We present two constraint languages whose grammar is shown in this section and compare them with each other.

Unification Constraints The type of constraints, which is created in [Gie01] are the simple unification constraints[48] which we state in Figure 3.13.

A constraint $C = (s_1 \equiv t_1 \wedge \ldots \wedge s_n \equiv t_n)$ is called satisfiable, if a unification can be found which serves all unification conditions $s_i \equiv t_i$. Unification and substitution are described in detail in [Fit96] and [BS01]. It is decidable, if such unification constraints are solvable or not. This fact makes them useful for practical applications.

Linear Unification Constraints Rümmer points out in [Rüm08], that unification constraints have shortcomings concerning applications that involve theories like arithmetic. For example, the constraint solver does not know that $2 = 1 + 1$ and therefore the constraint $X = 1 + 1 \wedge X = 2$ will not be solved. Therefore, Rümmer defines in [Rüm08] the language of linear unification constraints, which involves integer terms similar as defined in standard first-order logic and linear inequalities.

Figure 3.14 shows the grammar of this constraint language. $\mathsf{intTerm}$ is an integer valued term in Presburger Arithmetic [Pre91], which means it contains the literals $\ldots, -1, 0, 1, \ldots$ and the function symbols $+$ for addition and $-$ for subtraction. $\mathsf{othTerm}$ is a term of an arbitrary other type. The latter is included to handle also

[48] We quote the definition from [Rüm08].

non-integer terms, even though in a rudimentary way. In addition, a set of function symbols with integer or differently typed inputs is defined.

[Rüm08] defines the semantics of these constraints[49] similar to the semantics of the respective terms in first-order logic, for example in the definition of the semantics of terms, Definition 3.24.

Definition 3.45 (Semantics of Linear Unification Constraints)**.** For a vocabulary of function symbols and metavariables, a *structure* consists of

- a domain $\mathcal{D}_{\texttt{int}}$ (the mathematical integers) and $\mathcal{D}_{\texttt{oth}}$ (an arbitrary non-empty set), and
- an interpretation mapping $\mathcal{I}$ that maps each function symbol

  $$f : T_1 \times \cdots \times T_n \to T_0, \qquad T_i \in \{\texttt{int}, \texttt{oth}\}$$

  to a function

  $$\mathcal{I}(f) : \mathcal{I}(T_1) \times \cdots \times \mathcal{I}(T_n) \to \mathcal{I}(T_0)$$

  where we write $\mathcal{I}(\texttt{int}) := \mathcal{D}_{\texttt{int}}$ and $\mathcal{I}(\texttt{oth}) := \mathcal{D}_{\texttt{oth}}$. $\mathcal{I}$ is required to map the symbols $+, -$ and $\ldots, -1, 0, 1, \ldots$ to the corresponding operations on $\mathcal{D}_{\texttt{int}}$.

Given a vocabulary and a structure, a *variable assignment* is a mapping β that maps each metavariable $X : \texttt{int}$ to an individual $\beta(X) \in \mathcal{D}_{\texttt{int}}$ and each metavariable $Y : \texttt{oth}$ to an individual $\beta(Y) \in \mathcal{D}_{\texttt{oth}}$.

When a structure and a variable assignment are fixed, we can define how to evaluate terms and constraints. Evaluation is carried out by a mapping $\text{val}_{\mathcal{I},\beta}$ that is constructred inductively. The only interesting case is the third line where we define the meaning of $=$ to be equality on sets ($\mathcal{D}_{\texttt{int}}$ or $\mathcal{D}_{\texttt{oth}}$), and the meaning of $\neq, <, \leqslant$ to be the corresponding predicate on $\mathcal{D}_{\texttt{int}}$.

$$\begin{aligned}
\text{val}_{\mathcal{I},\beta}(X) &= \beta(X) \\
\text{val}_{f(t_1,\ldots,t_n)} &= \mathcal{I}(f)(\text{val}_{\mathcal{I},\beta}(t_1), \ldots, \text{val}_{\mathcal{I},\beta}(t_n)) \\
\text{val}_{\mathcal{I},\beta}(s * t) &= \begin{cases} \textit{true} & \text{val}_{\mathcal{I},\beta}(s) * \text{val}_{\mathcal{I},\beta}(t) \\ \textit{false} & \text{otherwise} \end{cases} \\
\text{val}_{\mathcal{I},\beta}(C \wedge D) &= \begin{cases} \textit{true} & \text{val}_{\mathcal{I},\beta}(C) = \textit{true} \text{ and } \text{val}_{\mathcal{I},\beta}(D) = \textit{true} \\ \textit{false} & \text{otherwise} \end{cases}
\end{aligned}$$

where $* \in \{=, \neq, <, \leqslant\}$.

Examples[50] for constraints in this language are the following.

[49] The following definition is taken from [Rüm08], Section 3.2.

[50] These examples are taken from [Rüm08].

Example 3.46 (Constraints)**.** We declare the functions $f, g : \texttt{int} \to \texttt{int}, h : \texttt{oth} \times \texttt{int} \to \texttt{oth}, c : \texttt{oth}$ and the metavariables $X, Y : \texttt{int}, A : \texttt{oth}$. Some linear unification constraints are

$$\begin{array}{ll} C_1 := X = 5 \wedge 2 = Y + 1 & C_2 := h(A, 2) = h(h(c, Y), Y + 1) \\ C_3 := f(1 + 2) = f(3) & C_4 := f(X) = g(Y) \\ C_5 := X < f(X) & C_6 := f(X) < f(Y) \\ C_7 := f(X) \leqslant f(Y) \wedge X \neq Y & C_8 := f(X) \leqslant f(Y) \wedge X < Y \end{array}$$

C_1, C_2, C_3 are satisfiable. C_4 is not satisfiable, because the ranges of $\mathcal{I}(f)$ and $\mathcal{I}(g)$ can be disjoint. C_5 is not satisfiable, because $\mathcal{I}(f)$ can be the identity. C_6 is not satisfiable, because $\mathcal{I}(f)$ can be constant. C_7 is satisfiable, because $\mathcal{I}(f)$ is either constant (and the values of X, Y can be chosen arbitrary but distinct), or there are numbers n_1, n_2 with $\mathcal{I}(f)(n_1) < \mathcal{I}(f)(n_2)$. C_8 is not satisfiable, because $\mathcal{I}(f)$ can be the function with $x \mapsto -x$.

Comparison of Unification and Linear Unification Constraints Linear unification constraints have the advantage of greater expressiveness, but they come with the drawback that the decidability of them is unknown so far. In general, any constraint language can be used for the purpose of incremental constraint solving, but there is always the trade-off between the expressiveness and the complexity of the decision procedure for the satisfiability of the constraints. For example, it would be possible to even use a non-linear constraint language, if only the satisfiability problem for this was not so hard.

The KeY prover, which we introduce in Section 3.5, creates linear unification constraints and uses an implementation[51] of Cooper's algorithm as external decision procedure for the constraints.

3.5 KeY

This work was carried out in the scope of the KeY Project, which is a joint project of the three European universities Chalmers Technical University in Gothenburg (Sweden), University of Karlsruhe (Germany) and University of Koblenz-Landau (Germany). In the project, a theorem prover named KeY is developed, which forms the base for our implementation (Chapter 6). In this section, we will give a brief overview over KeY and its features, especially those which were relevant for our work.

KeY is an interactive theorem prover, designed to reason about programs in dynamic logic. With KeY, we can verify that programs meet their specifications. The target programming language is Java Card[52], which is a fragment of full Java[53]. Specifications of programs can be written in JML, OCL or in KeY's own specification language in KeY-syntax.

[51] `http://www.cl.cam.ac.uk/users/jrh/atp/index.thml`

[52] See [Che00] for the specification of Java Card.

[53] See [GJSB05] for the specification of Java.

KeY covers all features of Java Card and a few additional ones of full Java. We will not give a full specification of Java Card, but state the relevant aspects for our work. KeY can deal among others with the following parts of programs:

- all built-in data types, except floating point numbers `double` and `float`,
- functions and arbitrary objects with attributes and methods,
- conditionals, loops
- recursive functions and object methods,
- arrays, lists, trees,
- side-effects of operations, assignments and conditions,
- exceptions.

Thus, KeY covers all features of the two languages which we examine, the While language and the Heap language, which we will introduce in Chapter 8.

KeY implements a calculus for Dynamic Logic which is called Calculus for KeY Java Card Dynamic Logic. This calculus covers all rules which we presented in Section 3.3, but of course includes much more rules to handle all the features which we stated in the preceding enumeration. The rules which we selected from the KeY calculus are the relevant ones for our work.

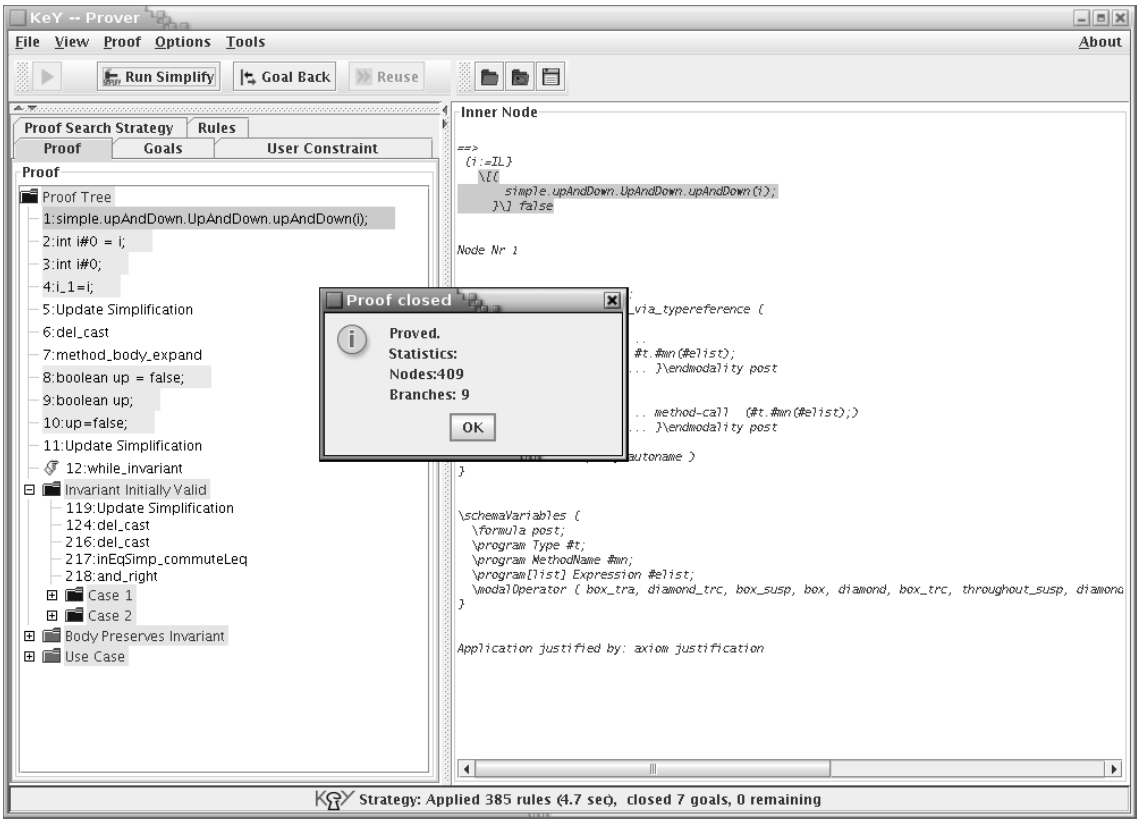
KeY is written in Java 1.4 and runs on all machines providing a suitable runtime environment.

Termination Analysis in KeY KeY is designed as a theorem prover to verify full specifications of programs. Specifications can (and mostly do) include the termination of a program. Thus, KeY provides of course ways to prove termination. For `while` loops, these are the invariant rules invRuleTerm and invRuleTermMod, which we presented in Section 3.3.2. Unfortunately, KeY does not provide the essential parts of this rule, the invariant *inv* and the ranking term t. Those have to be provided by the user.

KeY can also construct non-termination proofs. We will describe in Chapter 4, how non-termination proofs are made in dynamic logic. KeY works according to this description concerning non-termination proofs.

User Interaction KeY is designed as a semi-interactive theorem prover. Semi-interactive means, that most of the rule applications in the proof procedure are done automatically. The choice which rule to use on which sequent formula is made using powerful and well-adjusted heuristics. However, in some cases, additional information like invariants or ranking terms have to be provided. KeY comes with a graphical user interface, which provides ways for the user to insert those additional information. Figure 3.15 shows a screen shot of KeY's graphical interface.

In our work, we used KeY exclusively non-interactively. Of course, this restricts the power of KeY, but it was still sufficiently powerful to serve as a back end for our implementation. Although KeY is designed as an interactive prover, it provides an



Proof Tree
1:simple.upAndDown.UpAndDown.upAndDown(i);
2:int i#0 = i;
3:int i#0;
4:i_1=i;
5:Update Simplification
6:del_cast
7:method_body_expand
8:boolean up = false;
9:boolean up;
10:up=false;
11:Update Simplification
12:while_invariant
Invariant Initially Valid
119:Update Simplification
124:del_cast
216:del_cast
217:inEqSimp_commuteLeq
218:and_right
Case 1
Case 2
Body Preserves Invariant
Use Case

```
==>
 {i:=IL}
   \[{
       simple.upAndDown.UpAndDown.upAndDown(i);
     }\] false


Node Nr 1

                                          :
                                          _via_typereference {

                                           ..
                                          #t.#mn(#elist);
                                          ... }\endmodality post


                                          .. method-call  (#t.#mn(#elist);)
                                          ... }\endmodality post

                                          autoname )
}


\schemaVariables {
  \formula post;
  \program Type #t;
  \program MethodName #mn;
  \program[list] Expression #elist;
  \modalOperator { box_tra, diamond_trc, box_susp, box, diamond, box_trc, throughout_susp, diamond
}


Application justified by: axiom justification
```

Proved.
Statistics:
Nodes:409
Branches: 9

Strategy: Applied 385 rules (4.7 sec), closed 7 goals, 0 remaining

Figure 3.15: The User Interface of the KeY Prover

The user interface of KeY has basically two subwindows. In the window to the left, the proof tree is shown. The leafs are assigned different colors, depending on whether they are closed, closed by constraint or open. If the user clicks on one node in the proof tree, the node is shown in the right window with the information which sequent the node represents, which constraints are already found and which is the rule that is applied to transform this node's sequent to the next node's one (if any).

automatic mode, which we used in our implementation. For further technical details, see Chapter 6 about our implementation.

Chapter 4

Non-termination in Dynamic Logic

Using dynamic logic, we can specify all kinds of properties about programs. In particular, we can make statements about the termination behavior. In this chapter, we explain how to state the non-termination of a program in a dynamic-logic formula and how to prove it using the calculus.

4.1 Expressing Non-termination

Interestingly, for the reasoning about non-termination no means of reasoning about total correctness are necessary. The means for proving partial correctness are sufficient in this situation. In dynamic logic, partial correctness[1] of a program is expressed using the box modality $[\cdot]$. Typically, we have a formula of the form

$$[p]\varphi,$$

which expresses "If the program terminates, then it results in a state where φ holds". This formula does not require the program to actually terminate. If it does not terminate, the formula holds anyway.

If we merely want to express that the program does not terminate without any postcondition, then we simply form the formula

$$[p]\mathit{false}.$$

This way, the program only can fulfill the formula by not terminating[2].

The question how to express non-termination can also be approached from the other side. A total-correctness formula which demands that a program p *does* terminate and afterwards fulfill the property φ, looks like this.

$$\langle p\rangle\varphi$$

A formula which only says that the program p has to terminate (and not demands any other property), is this one.

$$\langle p\rangle\mathit{true}$$

[1] Partial correctness is seen in contrast to total correctness and described in Note 3.30.

[2] This is correct for While programs and but not completely correct for Heap programs. We will deal with this problem in Chapter 8, where we introduce Heap programs.

If we now want to express the *non-termination* of a program, we can simply negate this formula and obtain the following one.

$$\neg\langle p\rangle true$$

According to the definition of the semantics of dynamic-logic formulae (see Definition 3.29), this formula is equivalent to

$$[p]\neg true$$

and thus also to

$$[p] false,$$

which is the formula we presented in the first half of this section.

4.1.1 Non-termination versus Very Long Calculations

Program languages and logics can have different integer semantics. These semantics can differ in various things, but the relevant ones for our work are the treatment or occurrence of over- and underflows. Concerning under- or overflows, we can distinguish basically three different semantics.

1. *Classical Mathematical Semantics.* Integers, which are defined in the classical mathematical sense do not have any over- or underflows. A program which constantly increases an integer variable with this semantics will never reach an upper limit.
2. *Termination at Over- or Underflows.* These are semantics, where the set of integers has an upper and a lower limit and if a program hits these limits, it terminates with an exception.
3. *Cyclic Behavior at Over- or Underflows.* This type of semantics sets also an upper and a lower limit for integers, but programs which reach these limits do not crash. Instead, if for example the maximum integer is reached, they go on with the minimum integer and vice versa.

JAVA implements the third type of semantics for integers. The KEY prover provides actually all three kinds of semantics so that the user can choose by herself. For more information on integer arithmetic in KEY see [BS04] or Chapter 12 of [BHS07].

Whether a program terminates, is dependent on the integer arithmetic that is used. Consider the program GAUSS, which is shown in Figure 2.2.

If the program is started with a negative initial value for the variable `n`, then the value will be decreased in each iteration of the loop. Because the only termination condition for the loop is to encounter `n` to be zero, the decrease will go on and on and reach the minimum integer.

In the programs which implement the third type of semantics an underflow would occur in this situation This means that after another decrease of `n` the value will be the *maximum* integer. From there the decrease can continue until the value of 0 is reached and the loop actually terminates.

Thus a program with the third type of semantics and with an under- or overflow can terminate without an abrupt termination. Even if this is a long, but terminating loop, it is not what the programmer actually intended.

In the calculus, which we described in Section 3.3, we use an arithmetic for integers, which is not the arithmetic of JAVA. Instead, we use the first type of semantics, the idealized mathematical arithmetic for integers. This decision leads to the fact that long loops are identified as non-terminating loops, although they are actually terminating if we consider WHILE programs as a subset of JAVA programs.

We came to this decision, because we considered it as a long-term goal to improve the software development process by providing a method to find bugs in termination behavior. Because both, long loops and non-terminating loops, are bugs, they both should be detected by our method.

Another reason for choosing the mathematical definition of integers was that it makes reasoning about the programs easier. When using the mathematical semantics, we can apply well-studied properties of the mathematical integers.

4.1.2 Stating the Existence of Critical Inputs

There are two types of non-terminating programs. The programs of the first one do not terminate for *all* possible inputs and programs of the second one do not terminate for *some* inputs. In Section 4.1 we only described how to express the non-termination of a program of the first type, saying that the formula

$$[p] \mathit{false}$$

states that the program does not terminate for all possible inputs.

Identifying those programs is of course one of our goals (as described in Section 2.4), but identifying programs of the second type is more interesting, because those programs are more likely to be realistically written by a programmer.

If we want to express that a program does not terminate for *some* input values, we have to create a formula which quantifies existentially over all input variables. Let us denote $p(x_1, \ldots, x_n)$ as the program p with $x_1, \ldots, x_n$ being its input parameters. A formula which states that there are input values for which the program does not terminate is the following.

$$\exists x'_1 \, \ldots \exists x'_n \, \{x_1 = x'_1 \mid\mid \, \ldots \, \mid\mid x_n = x'_n\}[p(x_1, \ldots, x_n)] \, \mathit{false},$$

where $x'_1, \ldots, x'_n$ are logical variables of the same type as their corresponding program variables $x_1, \ldots, x_n$. We have to introduce those extra logical variables and go this long way round, because according to the specification of the logic, we cannot quantify over program variables (see Section 3.2.1).

4.2 Non-termination Proofs

According to Section 4.1.2, we can specify the non-termination of a program p for some inputs as the sequent

$$\Rightarrow \exists x'_1 \ldots \exists x'_n \; \{x_1 = x'_1 \mid\mid \ldots \mid\mid x_n = x'_n\}[p(x_1, \ldots, x_n)] \; \mathit{false}.$$

To prove these properties, a prover applies calculus rules to construct a proof. It first introduces metavariables for all existentially quantified variables. The formula then looks like this

$$\{x_1 = X_1 \mid\mid \ldots \mid\mid x_n = X_n\}[p(x_1, \ldots, x_n)] \; \mathit{false},$$

where $X_1, \ldots, X_n$ are the freshly introduced metavariables of the types which correspond to those of their pendants $x_1, \ldots, x_n$, respectively. This procedure of dealing with existentially quantified variables is described in Section 3.4 in detail. After the introduction of metavariables, the prover goes on with the proof construction the same way as usual.

In the construction of the proof, the program is symbolically executed. The most important rule application occurs at the point where the symbolic execution reaches the `while` statement of the program. At this point, an invariant rule (Section 3.3.2) is applied with φ being the formula *false*.

To have a closer look at these rules, we quote invRule[3] here again.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U} \mathit{inv}, \Delta \\ \mathit{inv}, \mathit{se} = \mathit{true} \Rightarrow [p]\mathit{inv} \\ \mathit{inv}, \mathit{se} = \mathit{false} \Rightarrow [\omega] \; \varphi\end{array}}{\Gamma \Rightarrow \mathcal{U}[\texttt{while (} \mathit{se} \texttt{) \{ } p \texttt{ \}} \; \omega]\varphi, \Delta} \; \textsf{invRule}$$

The formula φ occurs only in the third premiss of the rule, which is then equivalent to the formula

$$\mathit{inv} \Rightarrow \mathit{se}$$

provided that we leave out the rest of the program, ω, because in non-termination proofs it is not relevant anyway. This sequent formula says that the invariant *inv* has to imply the loop condition. If used in a non-termination proof, the invariant rule has three premises, which exactly match the three criteria of non-termination invariants in Section 2.5.

After the application of the invariant rule, the proof procedure continues the proof construction and hopefully can close the proof. Closing the proof usually requires to deal with the introduced metavariables. In Section 3.4, we presented the method of incremental proof closure by solving constraints. This method is applied here and it corresponds to the search for the critical inputs.

[3] In the non-termination proofs which our software constructed for the examples in this thesis, the rule invRuleMod was applied and not invRule. Nevertheless, we quote invRule here, because the properties which we want to talk about hold for this rule in the same way. We just quote this rule, because it is simpler and thus easier to read.

When applying this method of incremental closure, the metavariables are not actually instantiated with particular values, but instead solvable constraints are found, which describe requirements for the input variables to make the program not terminate. Thus, the search for the critical inputs of a program is transferred into constraint construction and the problem of checking the solvability of the constraints.

Solving the problem of finding the critical inputs with constraints has the advantage of that not only one particular tuple of critical inputs is found, but that the set of critical inputs is described more generally. This matches our third goal (see Section 2.4).

4.2.1 Interpretation of Successful Non-termination Proofs

After the successful construction of a non-termination proof, we receive two essential pieces of information. The first piece is the invariant which is used in the proof and the second is the set of constraints that is found to close the proof.

Meaning of Invariants. The invariant is an abstract description of a set of program states as we introduced it in Section 2.5. It describes the set of states which is never exited once it is entered in the loop.

Note, that the invariant can also contain metavariables. The information, for which values the metavariable can be substituted is derived in the constraints which are also a part of the solution of a non-termination analysis.

The set which corresponds to an invariant of a successful non-termination proof is not necessarily the only set with these properties nor is the invariant the most general description of such a set. The simpler the invariant is, the more general is the description of the set and the more critical inputs are usually covered.

Meaning of Constraints. The prover outputs the constraints for the introduced metavariables after completion of a proof. The constraints also describe a set of program states. The set which is described, is the set of start states from which the program's execution ends up in an infinite loop. By start state we mean the very first state which the program's execution visits and not necessarily the state which is reached in the execution right before the loop. These two states are the same, if the loop statement is the first of the program.

4.2.2 Interpretation of Failed Non-termination Proofs

There are situations in which the prover might not be able to close a proof. This can happen if for instance the invariant did not have the required properties (see Section 2.5). A failed proof is not completely worthless, because it contains valuable information about what went wrong in the proof and gives hints for how to solve these issues.

An open proof constraints at least one branch that has an open goal. Depending on in which branch the open goal occurs, we can draw different conclusions. We distinguish the branches according to from which premiss of the invariant rule they originate. The invariant rule invRule[4] has three premises which yield three corresponding proof subtrees.

1. *Open Goal in Init-branch.* The first premiss of the invariant rule states that the invariant has to be valid in the program state right before the entrance of the loop. If there is an open goal in a branch which originates from this premiss, it means that the invariant describes a set of states, of which none of its elements is reached with the entrance in the loop. If this is the case, the invariant is too strong, which means that the refinement process (see Section 2.6) has gone too far. In this case, there is no point in further refinement of the invariant candidate.
2. *Open Goal in Body-branch.* The second premiss of the invariant rule states that the invariant is valid after the execution of the loop body if it has been valid before. Open goals in this subtree indicate in which states the loop exits the set of states which the invariant describes. Information from these goals can be used to restrict the set of states further, to make it contain only states which are never left during the loop iterations.
3. *Open Goal in Use-case-branch.* The last premiss of the invariant rule contains the implication of the loop condition by the invariant in non-termination proofs. An open goal in this subtree means that the invariant is not sufficiently strong to imply the loop condition. The effect is that there might be loop iterations, where the invariant holds before and after the execution of the loop body (as it it is supposed to), but the loop terminates because the loop condition does not hold anymore although the invariant does. Thus, the information of an open goal here is useful to refine the invariant to make it imply the loop condition.

We finish this section by presenting an example program and discuss an open goal from the non-termination proof of it.

Example 4.1 (UpAndDown)**.** We show the example program's source code in Figure 4.1. It is a simple program with an input variable `i` and a local variable `up`. In case the initial value of `i` is in the range of 0 to 10, the behavior of the program makes the value of `i` go up to 10 and down to 0 over and over again in the iterations of the loop.

If we try to prove the non-termination of this program by an invariant which is too general, for example the invariant

$$i \geqslant 0,$$

the proof procedure fails in closing the proof and results among others in the open goal

$$i \geqslant 11 \Rightarrow,$$

which originates from the third premiss of the invariant rule, which is the one starting the use-case-branch. Therefore the open goal means, that the current variable is too

[4] See Figure 3.8 and its modification invRuleMod in Figure 3.10.

```
upAndDown(int i) {
  boolean up = false;
  while (0 <= i && i <= 10) {
    if (i == 10) {
      up = false;
    }
    if (i == 0) {
      up = true;
    }
    if (up) {
      i++;
    } else {
      i--;
    }
  }
}
```

Figure 4.1: UpAndDown
This program lets its input variable `i` run up to 10 and down to 0 over and over again.

general, because it does not imply the loop condition. That means in case of `i` being greater than 11, the loop terminates although the invariant still holds. We will examine this example further as Example 5.1.

4.3 Inverse Ranking Terms

In this section, we present and discuss an alternative approach to invariants. This approach is inspired by the ranking terms in partial correctness proofs.

In contrast to non-termination proofs, total correctness proofs require not only an invariant, but a ranking term. The ranking term is a term containing program variables, which can be evaluated to an element of a well-founded domain. The prover tries to prove that the value of the term strictly decreases in every iteration of the loop. In a well-founded domain, a chain of elements has always a minimum element. If the value of the ranking term decreases in every iteration, it will eventually reach the minimum element and cannot decrease any further. Thus, the loop terminates[5].

We considered adapting the idea of a ranking term to prove non-termination. The idea is to find an *inverse ranking term* t, which fulfills the following criteria:

1. t is evaluated to a domain $\mathcal{D}$, which is totally ordered by a relation $\prec$[6].
2. There is a fixed element $c_0 \in \mathcal{D}$ for which holds: If the value of t is greater than c_0 for a variable assignment β, then the loop condition *se* holds for the same variable assignment.

[5] See the description of the calculus rule invRuleTerm in Section 3.3.2 for further explanation.

[6] An example for such a domain are the natural numbers with the $<$ relation.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}t > c_0, \Delta \\ t > c_0 \Rightarrow se = \mathit{true} \\ c = t, se \Rightarrow [p]\; t \geqslant c\end{array}}{\Gamma \Rightarrow \mathcal{U}[\texttt{while (} se \texttt{) \{ } p \texttt{ \}}]\mathit{false}, \Delta}\ \mathsf{invRuleInverseRanking}$$

where c is a logical variable which did not occur in the proof so far.

Figure 4.2: Calculus Rule invRuleInverseRanking

```
nonLinearSimple(int i) {
  while (i*i > 9) {
    i++;
  }
}
```

Figure 4.3: NonLinearSimple
This program is an example for a non-linear operation in the loop condition. It does not terminate if `i` > 3.

3. The value of t is greater than c_0 in the program state from which the program enters the loop.
4. The value of t increases or stays stationary (but never decreases) with every iteration.

The second and third criterion ensure that the loop is entered. The second and fourth criterion make the loop execute over and over again, because the term never reaches c_0 and thus the loop condition always holds. Intuitively, the term is chosen to "move away" from the termination of the loop.

The calculus rule invRuleInverseRanking in Figure 4.2 expresses the idea of inverse ranking terms.

Example 4.2 (nonLinearSimple)**.** Figure 4.3 shows a program which does not terminate for all input values greater than 4.

We want to prove that this program does not terminate under the precondition of `i` > 4 and therefore we compose the proof obligation:

$$\texttt{i} > 4 \Rightarrow [\texttt{nonLinearSimple(i);}]\mathit{false}$$

Choosing $t = \texttt{i} \cdot \texttt{i}$ as inverse ranking term, the integers as totally ordered domain $\mathcal{D}$ and $c_0 = 9$, the application of the rule invRuleInverseRanking makes the proof branch into these three branches:

$$\frac{\begin{array}{c}\texttt{i} > 4 \Rightarrow \texttt{i} \cdot \texttt{i} > 9 \\ \texttt{i} \cdot \texttt{i} > 9 \Rightarrow (\texttt{i} \cdot \texttt{i} > 9) = \mathit{true} \\ \texttt{i} \cdot \texttt{i} = c, \texttt{i} \cdot \texttt{i} > 9 \Rightarrow [\texttt{i++}]\; \texttt{i} \cdot \texttt{i} \geqslant c\end{array}}{\texttt{i} > 4 \Rightarrow [\texttt{while (} i * i > 9 \texttt{) \{ i++; \}}]\; \mathit{false}}\ \mathsf{invRuleInverseRanking}$$

We can easily close the branches using calculus rules for arithmetic and symbolic execution and thus prove the non-termination of the program.

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

Figure 4.4: AlternatingIncr
This program alternatingly increases and decreases its input variable `i`. In total the value is increased more than it is decreased, which leads to an endless loop if the initial value of `i` is positive.

A closer look at the four criteria reveals that they are actually stricter than they need to be. It is sufficient to prove that t never reaches c_0 instead of demanding that it never decreases. Thus a program, where t decreases within reason might not terminate although t misses the fourth criterion. Have a look at the following example.

Example 4.3 (AlternatingIncr)**.** In the program, whose code we show in Figure 4.4, the value of the input variable `i` is decreased, respectively increased, when its value is even, respectively odd. If the program is started with a positive value, it does not terminate.

Intuitively, we would pick $t =$ `i` as inverse ranking term, but since the value of `i` is alternatingly decreased and increased, t would not fulfill the fourth criterion and thus we would not be able to prove the non-termination of the program, although it actually does not terminate.

An alternative calculus rule would weaken the third premiss and look like this.

$$\frac{\begin{array}{c}\Gamma \Rightarrow \mathcal{U}t > c_0, \Delta \\ t > c_0 \Rightarrow se = \mathit{true} \\ t > c_0, se \Rightarrow [p]\; t > c_0\end{array}}{\Gamma \Rightarrow \mathcal{U}\langle \texttt{while (}\ se\ \texttt{)}\ \{\ p\ \}\ \omega\rangle \mathit{false}, \Delta}\ \textsf{invRuleInverseRankingSoft}$$

With this rule we would be able to prove the non-termination of Example 4.3.

Equivalence to the Invariant Rule The rule invRuleInverseRankingSoft actually is the invariant rule with $t > c_0$ as invariant *inv*. Thus, the approach of inverse ranking terms is subsumed by the approach of using invariants for non-termination proofs.

4.4 Different Kinds of Invariants

Invariants are not only used in non-termination proofs. The calculus rules of Figures 3.8 and 3.10 can also be applied in proofs where other properties of programs are proven. In this section we refer to invariants in those proofs as *correctness invariants* to distinguish them from non-termination invariants (see Section 2.5).

These two kinds of invariants have some common properties, but there are also some essential differences. Invariants of both kinds hold before the loop execution starts and they *are* invariants, which means they are preserved during execution of the loop body.

The essential difference is that non-termination invariants imply the loop condition. This is the crucial point for proving non-termination, which means that correctness invariants must not have this property. Instead, they are used to conclude properties about the program state after the loop execution. Because non-terminating programs never reach the program statements after the loop, there is no point in proving any properties of the programs execution after the loop. Thus, non-termination invariants do not have any further purpose after the loop, whereas the correctness invariants have.

In our work, we examined a number of proofs and from this experience we conclude properties concerning the form of invariants. Non-termination invariants tend be much simpler formulae than correctness invariants are. This is due to the fact that non-termination invariants just describe sets of states, whereas correctness invariants have to capture information about the manipulation that is done in the loop in order to prove properties about the program state after the loop. Thus, the task of invariant generation for non-termination proofs is easier than the invariant generation of correctness proofs.

Chapter 5

Algorithm

We designed an algorithm which automatically determines, whether a program terminates. It does this by incrementally generating non-termination invariants. We will start the description by introducing the general idea of the algorithm in the first section, then show an example as illustration and then give a more detailed insight to the inner workings of the algorithm. The algorithm is illustrated by examples throughout the chapter.

5.1 General Idea of the Algorithm

The algorithm can be subdivided into two components: an invariant generator and a theorem prover for dynamic logic. These components work together in the following manner.

The input of the algorithm is a program source of the program p containing a non-nested `while` loop[1] which is to be analyzed for non-termination. $x_1, \ldots, x_n$ are the input parameters of the program. The algorithm goes through the following steps.

Initialization

1. The program $p(x_1, \ldots, x_n)$ is inserted in a dynamic-logic formula φ, which existentially quantifies over all possible program inputs and states that there are program inputs that cause the program to not terminate (see Section 4.1.2).

$$\varphi \equiv \exists x'_1 \ldots \exists x'_n \; \{x_1 = x'_1 \;||\; \ldots \;||\; x_n = x'_n\}[p(x_1, \ldots, x_n)] \; \mathit{false}$$

2. The formula φ is given to the theorem prover. The proof procedure is invoked and constructs a proof tree in which the program is symbolically executed until the execution reaches the loop.
3. The variable for the invariant candidate is set initially to the formula $\mathit{inv}_1 \equiv \mathit{true}$.

[1] This very version of the algorithm works on non-nested loops only, but it can be applied to nested loops under certain conditions. We describe the generalisation of the algorithm to nested loops in Section 5.4.

Iteration

4. The proof procedure applies the invariant rule invRuleMod (Figure 3.10), which we described in Section 3.3.2. The invariant *inv* which is used in the invariant rule's first application is the invariant candidate inv_i with i being dependent on the number of iteration (in the first one it is $inv_1 \equiv true$, which was set in step 3).
5. The proof procedure keeps on constructing the proof as far as possible without human interaction.
6. If the proof procedure can close the proof, the algorithm terminates with the result that the program does not terminate. If the proof cannot be closed, the algorithm extracts the open goals from the proof and hands them over to the invariant generator.
7. The invariant generator extracts information from the formulae in the open goals to refine the invariant candidate, which was used in the application of the invariant rule in step 4. That means, the generator creates[2] several new invariant candidates with the help of the open goals and the current invariant candidate inv_i.

The algorithm repeats step 4 to 7 iteratively, using one of the newly created invariant candidates[3] in the invariant rule. The iterations are carried out until one of these events occurs: a non-termination invariant is found, which means that the proof was closed with the help of the invariant candidate, the algorithm runs out of new invariant candidates or a maximum number of iterations is reached.

The algorithm is sketched in Figure 5.1 and given in a condensed way as JAVA-like code in Figure 5.2. We talk about the inner workings of the algorithm in more detail in Section 5.2.

Results of the Algorithm The algorithm has three ways to terminate. If it terminates by closing a proof, it outputs three pieces of information. It says, that the input program does not terminate, returns the invariant that is used in the application of the invariant rule and states the constraints, which were left in the open goals and which the constraint solver identified as solvable[4].

The second way the algorithm can terminate is to run out of invariant candidates. This can happen if all invariant candidates that have been created from the information in the open goals have been tried in a proof attempt without finding an invariant which leads to a closed proof. This is usually the case when the invariant which is needed to make the proof close is too complex to be created by any of our creation methods.

The last way of the algorithm to terminate is when the number of iterations reaches an upper limit. This limit is necessary, because we easily constructed pro-

[2] For the details of the invariant creation step see Section 5.2.

[3] Which invariant candidate is chosen is described in Section 5.2.

[4] For detailed description of the application of the constraint solver, see Section 3.4. We talked about the interpretation of the invariant and the constraints in Section 4.2.1.

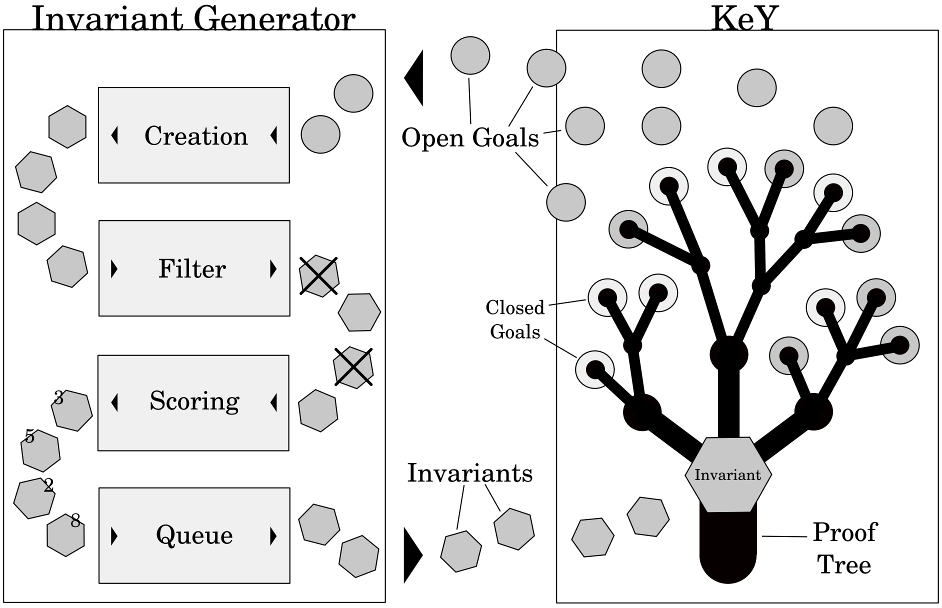


Figure 5.1: Algorithm Sketch
This is a sketch of the algorithm. The open goals of a failed proof attempt are taken to create new invariant candidates for further proof attempts.

grams whose open proofs lead to more and more new invariant candidates without ever finding one that closes the proof. The introduction of the limit ensures the termination of our algorithm itself.

We enrich the presentation of the algorithm by some examples. At this point of the description not every single step is supposed to be fully understandable to the reader, but we introduce an example here to give a first insight.

Example 5.1 (UpAndDown)**.** We apply our algorithm to the program UpAndDown, which we introduced in Chapter 4 as Example 4.1.

The only invariant up to equivalence that fulfils the three criteria for a non-termination invariant (Section 2.5), is the formula

$$0 \leqslant i \wedge i \leqslant 10.$$

We apply the algorithm on this example by going through it step by step.

Initialization

1. The first step is to form the formula which expresses the non-termination of at least one possible input. For the case of this program, the formula is

$$\varphi \equiv \exists i' \; \{\texttt{i} = i'\}[\texttt{boolean up = false; while ...}] \, \mathit{false}$$

```
proveNontermination(Program p) {
  // interface to the theorem prover
  TheoremProver tp = ...
  // invariant generator
  InvariantGenerator ig = ...
  // formula stating the non-termination of p
  Formula phi = formProofObligation(p);
  // number of iterations
  int noIt = 0;
  // maximum number of iterations
  int maxIt = [...];
  // queue of invariant, initialized with formula "true"
  InvariantQueue q;
  q.add(new Formula(true));
  // current invariant of the iteration
  Formula curInv;
  while (noIt < maxIt && !q.isEmpty()) {
    noIt++;
    // pick next invariant from the queue
    curInv = q.getNext();
    // construct proof
    Proof proof = tp.prove(phi, curInv);
    // check for result
    if (proof.isClosed()) {
      print "Success: The program does not terminate.";
    }
    // new invariants are generated and added to the queue
    q.add(ig.generateNewInvs(proof.getOpenGoals(), curInv));
  }
  // check for kind of failure
  if (q.isEmpty()) {
    return "Failed: No more invariants to try.";
  }
  return "Failed: Maximum number of iterations reached.";
}
```

Figure 5.2: Algorithm in Java-like code.

2. The second step of the algorithm is to let the theorem prover construct a proof up to the point where the symbolic execution reaches the `while` loop. As first step in the proof construction, the quantification of the variable i' is replaced by the introduction of the metavariable I. This proof stub is to be read from bottom to top.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Rightarrow \{\mathtt{i} := I \mathrel{||} \mathtt{up} := \mathtt{false}\}[\mathtt{while}\ldots]\, \mathit{false}}}{\Rightarrow \{\mathtt{i} := I\}[\mathtt{boolean\ up\ =\ false;\ while}\ldots]\, \mathit{false}}}{\Rightarrow \exists i'\{\mathtt{i} := i'\}[\mathtt{boolean\ up\ =\ false;\ while}\ldots]\, \mathit{false}}$$

3. The initial invariant candidate is set to $\mathit{inv}_1 \equiv \mathit{true}$.

Iteration no. 1

4. At this point in the proof, the invariant rule invRuleMod is applied using as invariant candidate the initial one, $\mathit{inv}_1 \equiv \mathit{true}$. The reduced context is empty. The proof branches threefold.

$$\dfrac{\begin{array}{c}\Rightarrow \{\mathtt{i} := I \mathrel{||} \mathtt{up} := \mathtt{false}\}\ \mathit{true} \\ \mathcal{A}\ \mathit{true}, \mathcal{A}\ 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant 10 \Rightarrow \mathcal{A}\ [\mathtt{if\ (i\ ==\ 10)}\ldots]\ \mathit{true} \\ \mathcal{A}\ \mathit{true} \Rightarrow \mathcal{A}\ 0 \leqslant \mathtt{i} \wedge \mathtt{i} \leqslant 10\end{array}}{\Rightarrow \{\mathtt{i} := I \mathrel{||} \mathtt{up} := \mathtt{false}\}[\mathtt{while}\ldots]\, \mathit{false}}\ \textsf{invRuleMod}$$

with

$$\mathcal{A} = \{\mathtt{i} := i^* \mathrel{||} \mathtt{up} := \mathit{up}^*\ \}.$$

5. The theorem prover goes on constructing the proof as far as possible. Due to space limitations, we cannot state the whole proof here, but Figure 5.3, 5.4 and 5.5 show a slightly shortened version of the proof tree.

6. Because the proof tree is not closed, we move on to step 6 of the algorithm. The open goals of this proof are:

$$\mathtt{i} \geqslant 11 \Rightarrow \quad \text{and} \quad \mathtt{i} \leqslant -1 \Rightarrow$$

7. The invariant generator examines the open goals and produces the following new invariant candidates. We will explain the exact details of this procedure in the succeeding section.

$$\mathit{inv}_2 \equiv \mathit{true} \wedge \mathtt{i} < 11 \quad \text{and} \quad \mathit{inv}_3 \equiv \mathit{true} \wedge \mathtt{i} > -1$$

The new invariant candidates are inserted in a queue and the first candidate of the queue is taken as invariant for a new iteration.

$$\dfrac{\dfrac{\dfrac{\text{branch}_0 \quad \text{branch}_1 \quad \text{branch}_2}{\Rightarrow \{\mathtt{i} := I \mid\mid \mathtt{up} := \mathtt{false}\}[\mathtt{while}\ldots]\mathit{false}}}{\Rightarrow \{\mathtt{i} := I\}[\mathtt{boolean\ up\ =\ false;\ while}\ldots]\mathit{false}}}{\Rightarrow \exists i^l\{\mathtt{i} := i^l\}[\mathtt{boolean\ up\ =\ false;\ while}\ldots]\mathit{false}}$$

$$\dfrac{\dfrac{\dfrac{}{\Rightarrow \mathit{true}}\ \mathsf{closeTrue}}{\Rightarrow \{\mathtt{i} := I \mid\mid \mathtt{up} := \mathtt{false}\}\mathit{true}}}{\text{branch}_0}$$

$$\dfrac{\dfrac{\dfrac{\Rightarrow 0 \leqslant i^* \quad \Rightarrow i^* \leqslant 10}{\mathit{true} \Rightarrow 0 \leqslant i^* \land i^* \leqslant 10}}{\{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathit{up}^*\}\ \mathit{true} \Rightarrow \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathit{up}^*\}\ 0 \leqslant \mathtt{i} \land \mathtt{i} \leqslant 10}}{\text{branch}_2}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* = 10,\\ i^* = 0, \mathit{false} \Rightarrow \ldots\end{array}}\ \mathsf{closeFalse}}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* = 10,\\ i^* = 0, 0 = 10 \Rightarrow \ldots\end{array}}}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* = 0\\ \Rightarrow \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathtt{false}\}\\ [\mathtt{up\ =\ true;}\ \ \ldots]\,\mathit{true}\end{array}} \quad \dfrac{\text{branch}_{1010} \quad \text{branch}_{1011}}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* \neq 0\\ \Rightarrow \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathtt{false}\}\\ [\mathtt{if\ (up)}\ldots]\,\mathit{true}\end{array}}}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* = 10\\ \Rightarrow \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathtt{false}\}\\ [\mathtt{if\ (i\ ==\ 0)}\ldots]\,\mathit{true}\end{array}}}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* = 10\\ \Rightarrow \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathit{up}^*\}\\ [\mathtt{up\ =\ false;}\ldots]\,\mathit{true}\end{array}} \quad \text{branch}_{11}}{\dfrac{\begin{array}{c}\{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathit{up}^*\}\ \mathit{true}, \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathit{up}^*\}\ 0 \leqslant \mathtt{i} \land \mathtt{i} \leqslant 10\\ \Rightarrow \{\mathtt{i} = i^* \mid\mid \mathtt{up} = \mathit{up}^*\}\,[\mathtt{if\ (i\ ==\ 10)}\ldots]\,\mathit{true}\end{array}}{\text{branch}_1}}$$

Figure 5.3: Proof Tree of UpAndDown, Part 1 of 3
This is the proof tree which the theorem prover produces for the (failed) non-termination proof of the program UpAndDown in the first iteration of the algorithm.

$$\dfrac{\dfrac{\dfrac{}{0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* \neq 0, \mathit{false} \Rightarrow \ldots}\ \textsf{closeFalse}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* \neq 0, \texttt{false} = \texttt{true} \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \texttt{false}\}\, [\texttt{i++}\ldots]\, \mathit{true} \end{array}}}{\text{branch}_{1010}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* \neq 0, \Rightarrow \mathit{true}}\ \textsf{closeTrue}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* \neq 0, \\ \Rightarrow \{\texttt{i} = i^* - 1 \mid\mid \texttt{up} = \texttt{false}\}\mathit{true} \end{array}}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* = 10, i^* \neq 0, \texttt{false} = \texttt{false} \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \texttt{false}\}\, [\texttt{i--;}\ldots]\, \mathit{true} \end{array}}}{\text{branch}_{1011}}$$

$$\dfrac{\dfrac{\text{branch}_{110} \quad \text{branch}_{111}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10 \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \mathit{up}^*\}\, [\texttt{if (i = 0)}\ldots]\, \mathit{true} \end{array}}}{\text{branch}_{11}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, \\ i^* = 0 \Rightarrow \mathit{true} \end{array}}\ \textsf{closeTrue}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, \\ i^* \neq 10, i^* = 0 \\ \Rightarrow \{\texttt{i} = i^* + 1 \mid\mid \texttt{up} = \texttt{true}\} \\ \mathit{true} \end{array}}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, \\ i^* = 0, \texttt{true} = \texttt{true} \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \texttt{true}\} \\ [\texttt{i++;}\ \ldots]\, \mathit{true} \end{array}} \quad \dfrac{\dfrac{}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, \\ i^* = 0, \mathit{false} \\ \Rightarrow \{\texttt{i} = i^* - 1 \mid\mid \texttt{up} = \texttt{true}\}\mathit{true} \end{array}}\ \textsf{closeFalse}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, \\ i^* = 0, \texttt{true} = \texttt{false} \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \texttt{true}\} \\ [\texttt{i--;}\ldots]\, \mathit{true} \end{array}}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, i^* = 0 \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \texttt{true}\}\, [\texttt{if (up)}\ \ldots]\, \mathit{true} \end{array}}}{\begin{array}{c} 0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, i^* = 0 \\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = \mathit{up}^*\}\, [\texttt{up = true;}\ldots]\, \mathit{true} \end{array}}}{\text{branch}_{110}}$$

Figure 5.4: Proof Tree of UpAndDown, Part 2 of 3
See caption of Figure 5.3.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10,\\ i^* \neq 0, up^* = \texttt{true}\\ \Rightarrow \mathit{true}\end{array}}\,\textsf{closeTrue}
}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10,\\ i^* \neq 0, up^* = \texttt{true}\\ \Rightarrow \{\texttt{i} = i^* + 1 \mid\mid \texttt{up} = up^*\}\mathit{true}\end{array}}
}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10,\\ i^* \neq 0, up^* = \texttt{true}\\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = up^*\}\\ [\texttt{i++;}\ldots]\,\mathit{true}\end{array}}
\qquad
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10,\\ i^* \neq 0, up^* = \texttt{false}\\ \Rightarrow \mathit{true}\end{array}}\,\textsf{closeTrue}
}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10,\\ i^* \neq 0, up^* = \texttt{false}\\ \Rightarrow \{\texttt{i} = i^* - 1 \mid\mid \texttt{up} = up^*\}\mathit{true}\end{array}}
}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10,\\ i^* \neq 0, up^* = \texttt{false}\\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = up^*\}\\ [\texttt{i--;}\ldots]\,\mathit{true}\end{array}}
}{\begin{array}{c}0 \leqslant i^*, i^* \leqslant 10, i^* \neq 10, i^* \neq 0\\ \Rightarrow \{\texttt{i} = i^* \mid\mid \texttt{up} = up^*\}\,[\texttt{if (up)}\ldots]\,\mathit{true}\end{array}}
}{\text{branch}_{111}}
$$

Figure 5.5: Proof Tree of UpAndDown, Part 3 of 3
See caption of Figure 5.3.

Iteration no. 2

3.-5. The theorem prover tries to prove φ using the invariant candidate inv_2. The proof cannot be closed and the open goals are:

$$I \geqslant 11 \Rightarrow \quad \text{and} \quad \texttt{i} \leqslant -1 \Rightarrow$$

6. The invariant generator refines invariant candidate inv_2 with the help of the open goals and we obtain the new invariant candidates inv_4 and inv_5, which are added to the queue

$$inv_4 \equiv \mathit{true} \wedge \texttt{i} < 11 \wedge \texttt{i} > -1$$
$$inv_5 \equiv \mathit{true} \wedge \texttt{i} < 11 \wedge I < 11$$

Iteration no. 3

3.-5. The next invariant in the queue is inv_3. The proof which uses inv_3 in the invariant rule results in the open goals:

$$I \leqslant -1 \Rightarrow \quad \text{and}\ \texttt{i} \geqslant 11$$

6. The invariant generator refines the invariant candidate inv_3 and produces the new invariant candidates inv_6 and inv_7.

$$inv_6 \equiv \mathit{true} \wedge \texttt{i} > -1 \wedge I > -1$$

$$inv_7 \equiv \mathit{true} \wedge \texttt{i} > -1 \wedge \texttt{i} < 11$$

Because inv_7 is equivalent to inv_4, it is dismissed and only inv_6 is added to the queue.

Iteration no. 4

3.-5. The next invariant candidate in the queue is $inv_4 = true \wedge \texttt{i} < 11 \wedge \texttt{i} > -1$. The theorem prover uses it to construct another proof and this time, the proof can be closed with the constraints:

$$-1 < I \text{ and } I < 11$$

Interpretation of the Result The result is to be interpreted as follows. The constraint says that the initial value of `i`, which is represented by the metavariable I, must be between -1 and 11 in order to enter the infinite loop.

The invariant itself describes the set of values of `i`, which is never left in the execution of the loop; in this case it is stated by the same conditions as the constraints for the initial value. The invariant says that the value of `i` is always between -1 and 11.

In summary, the algorithm terminates with a positive result, announcing that the program does not terminate for the critical input values between -1 and 11.

5.2 Inner Workings of the Invariant Generator

The invariant generator receives as input the open goals of a failed proof attempt and outputs a new invariant candidate for the next iteration of the algorithm. The process of invariant generation goes through four phases, which we will describe in more detail in the succeeding sections and illustrate in Figure 5.1:

- *Creation.* In this phase, the incoming open goals are examined and a new invariant candidate is created by the refinement of the current invariant candidate using the information of the open goals.
- *Filtering.* Because some of the newly created invariant candidates are useless for the proof of non-termination, they are filtered out by various criteria.
- *Scoring.* Usually, in each iteration several new invariant candidates are created and thus lead to the question of which of the new invariant candidates to try in the next iteration. This problem is solved by assigning a score to each new invariant candidate and queuing it according to the score.
- *Queuing.* This phase is merely queuing the new invariant candidates according to the score they were assigned in the preceding phase.

5.2.1 Creation of Invariants Candidates

The creation step of the algorithm receives a set of open goals $g_1, \ldots, g_n$ as input, which are sequent formulae of the form $\Gamma_i \Rightarrow \Delta_i$ for each goal g_i, where Γ_i and Δ_i are sets of formulae. There are several ways to extract information from open goals.

In general we can say, that each creation method examines the formulae of the open goals and creates a new formula ρ from it, which is called invariant fragment.

This invariant fragment is then conjunctively added to the old invariant candidate to obtain the new invariant candidate.

$$\mathit{inv}_{\text{new}} \equiv \mathit{inv}_{\text{old}} \wedge \rho$$

In the version of the algorithm which we present here, all methods of invariant generation are applied in each iteration, but of course in an actual implementation the user might choose the creation methods she considers as most effective. In the Example 4.1, we used only the creation method no 2.

1. Add a Formula of the Succedent. A formulae $\varphi \in \Delta_i$ in the succedent states a situation where there is a problem with the non-termination proof when φ does not hold. Most often that means that in this situation the loop actually terminates[5]. We would like to exclude this situation in the invariant and therefore we add φ conjunctively to the old invariant candidate:

$$\mathit{inv}_{\text{new}} \equiv \mathit{inv}_{\text{old}} \wedge \varphi$$

Example 5.2. Assume we are dealing with a program, in whose termination proof we used the invariant

$$\mathit{true} \wedge \mathtt{k} > 0.$$

The application of the creation method of adding a formula of the succedent on the open goal

$$\mathtt{i} = 5 \Rightarrow \mathtt{i} \geqslant 3, \mathtt{k} \geqslant \mathtt{i}$$

yields these two new invariants:

$$\begin{gathered}\mathit{true} \wedge \mathtt{k} > 0 \wedge \mathtt{i} \geqslant 3\\ \mathit{true} \wedge \mathtt{k} > 0 \wedge \mathtt{k} \geqslant \mathtt{i}\end{gathered}$$

2. Negate and Add a Formula of the Antecedent A formula $\varphi \in \Gamma_i$ in the antecedent means that there is a problem with the non-termination in the situation where φ holds. Here, the same idea applies as for formulae in the succedent, but in this case we have to negate it before we add it to the old invariant conjunctively:

$$\mathit{inv}_{\text{new}} \equiv \mathit{inv}_{\text{old}} \wedge \neg\varphi$$

Example 5.3. In iteration no 1 of Example 4.1, we obtain the open goals

$$i \geqslant 11 \Rightarrow \text{ and } \mathtt{i} \leqslant -1 \Rightarrow$$

which are transformed into the following invariant candidates by the creation method just described.

$$\begin{gathered}\mathit{inv}_1 \equiv \mathit{true} \wedge \mathtt{i} < 11\\ \mathit{inv}_2 \equiv \mathit{true} \wedge \mathtt{i} > -1\end{gathered}$$

[5] See Section 4.2.1 for the interpretation of open goals.

3. Replacement of Equality in a Formula in the Antecedent by an Inequality. If there is a formula in the antecedent of the open goals that contains an equality as topmost operation, i. e. $a = b \in \Gamma_i$, then instead of just negating the formula, we replace the equality by two inequalities and add them to the old invariant candidate. Thus the newly created invariant candidates with this creation methods are:

$$inv_{\text{new}} \equiv inv_{\text{old}} \wedge a > b \quad \text{and} \quad inv_{\text{new}} \equiv inv_{\text{old}} \wedge a < b$$

Example 5.4 (Gauss)**.** Figure 2.2 shows a simple program which is supposed to sum up all integers between 0 and the value of the variable `n`. This works well unless the initial value of `n` is negative. In this case, the value of the variable is decreased more and more until it reaches an underflow.

Applying the algorithm to this example yields in the first iteration the only open goal

$$\mathtt{n} = 0 \Rightarrow$$

Applying the creation method no 2 on this formula does not have the desired effect, because the application of the invariant rule with the invariant candidate $inv_2 \equiv true \wedge \mathtt{n} \neq 0$ yields the open goal

$$\mathtt{n} = 1 \Rightarrow$$

Going on adding the formulae $\mathtt{n} \neq 1, \mathtt{n} \neq 2, \ldots$ will not be an efficient way to construct a non-termination invariant. Therefore instead of adding $\mathtt{n} \neq 0$, we apply creation method no 3 and add the inequalities $\mathtt{n} < 0$ and $n > 0$ to the invariant and obtain the new invariant candidates:

$$true \wedge \mathtt{n} < 0 \quad \text{and} \quad true \wedge \mathtt{n} > 0$$

Of which the first one leads to a closed proof and thus shows that the program does not terminate for negative input values.

4. Introducing Metavariables in Inequalities A more aggressive creation method is to introduce unspecified lower or upper bounds for each occurring term in an open goal. This is done by introduction of metavariables in the invariant candidate.

This creation method works in two phases. In a first phase, all integer terms $t_1, \ldots, t_m$ which occur in the open goals $\psi_1, \ldots, \psi_n$ are extracted. In the second phase for each term t_i, two new formulae are created which state that this term is greater respectively less than a metavariable L_i respectively U_i. So for each t_i with $i = 1, \ldots, m$, we obtain the new invariant candidates:

$$inv_{\text{new}_1} \equiv inv_{\text{old}} \wedge t_i > L_i \quad \text{and} \quad inv_{\text{new}_1} \equiv inv_{\text{old}} \wedge t_i < U_i$$

The metavariables have to be fresh, which means they must not have occurred in the proof so far. By introduction of the metavariables we say that there is a lower

```
while (i > 0) {
  if (i > 5) {
    i++;
  } else {
    i--;
  }
}
```

Figure 5.6: UpOrDown
This example program increases the value of the variable `i` if it is greater than 5 and decreases it otherwise. The program does not terminate for all input values greater than 5.

Iteration no	Current Invariant	Open Goals
1	*true*	$\mathtt{i} \leqslant 0 \Rightarrow$
2	*true* $\wedge\ i > 0$	$\mathtt{i} = 1 \Rightarrow$
3	*true* $\wedge\ i > 0 \wedge i > 1$	$\mathtt{i} = 2 \Rightarrow$
⋮	...	...
7	*true* $\wedge\ i > 0 \wedge i > 1 \wedge \ldots \wedge i > 5$	none

Figure 5.7: Refinement of the Invariant for UpOrDown
This table shows the iteration steps of the algorithm applied on the example program UpOrDown. A non-termination invariant is found in the 7th iteration.

respectively upper bound, but we do not specify which one. The metavariables which we introduce this way are treated in the proof procedure as if they came from existentially quantified variables in the first place[6]. We exploit the concept of metavariables to deal with fixed but not yet specified values in proofs to introduce such unknown bounds.

Example 5.5 (UpOrDown)**.** The program shown in Figure 5.6 contains a loop that runs for positive values of the variable `i`.

The application of the algorithm can take the development[7] which is shown in Figure 5.7, if one does only apply the creation methods 1 and 3.

The invariant that is found in the 7th iteration leads to a closed proof, but it is an unnecessary complex invariant and if the `if`-condition contained a higher number than 5, the invariant would have grown even more complex.

When applying the creation method which we just described, we form the invariant fragment $i > M$ instead of $i > 0$, where M is a metavariable that has not occurred in the proof so far. That means, instead of saying that 0 is a lower bound for i, we say that there is a lower bound for i, but we do not specify its value yet.

With this creation method the second iteration of the algorithm would apply the

[6] Metavariables were first introduced to the calculus to handle existentially quantified variables. The general way to deal with metavariables in proofs is described in Section 3.4.

[7] We say *can*, because the order of the application of the creation methods is not specified in the description of the algorithm and thus might vary in different implementations. Besides that, we left out some iterations that do not help to find the desired invariant in this case.

invariant rule using the invariant candidate:

$$inv_0 \equiv true \wedge i > M$$

With this candidate, the proof can be closed with the following constraints for the metavariables:

$$M < I \wedge -1 < M \wedge 4 < M$$

We can interpret this information as follows (Section 4.2.1). The program does not terminate for all input values greater than the lower bound M, where M has to be greater than 4. Or in other words: The program does not terminate for all program inputs greater than 5.

For the introduction of heuristics for the scoring of invariant candidates, we need to distinguish the different types of metavariables. We call them start-state metavariables and invariant metavariables. Those two types are the only kind of metavariables which can occur in proofs in our algorithm.

Definition 5.6 (Start-state Metavariables and Invariant Metavariables)**.** *Start-state Metavariables* are metavariables which are introduced when replacing the existantially quantified input variables of the program by metavariables. *Invariant metavariables* are metavariables which are freshly introduced in creation method no 4 (Introducing Metavariables in Inequalities).

5. Adding Formulae Disjunctively. Another approach is not to add the newly created formula conjunctively, but to take a pair of newly created formulae, combine those disjunctively and add the disjunction conjunctively to the old invariant candidate. Thus, this creation method can be considered as a meta creation method, because it takes the invariant fragments that have been created by the methods 1 to 4 and forms new invariants using them.

Assume, using creation methods 1 to 4 we obtain the formulae $\rho_1, \ldots, \rho_p$ as invariant fragments. For each pair ρ_{i_k}, ρ_{i_l} with $1 \leqslant k \leqslant p, 1 \leqslant l \leqslant p, k < l$ we form the disjunction $\rho_{i_k} \vee \rho_{i_l}$ and add it to the invariant:

$$inv_{\text{new}} \equiv inv_{\text{old}} \wedge (\rho_{i_k} \vee \rho_{i_l})$$

The idea behind this creation method is to be able to handle non-termination proofs of programs that do not have invariants that can be expressed by a conjunction. Invariants which can be described as a conjunction are called *convex invariants*. The idea can be broadened by forming disjunctions of more than two invariant fragments, but for simplicity reasons, we limit the number of fragments to two.

Example 5.7 (AlternDivWide)**.** Figure 5.8 shows a program, whose invariant can be described with a disjunction much more easily than with conjunctions only. Over the iterations of the loop the absolute value of the variable `i` increases, but in every iteration the sign of the value is flipped.

```
alternDivWidening(int i) {
  int w = 5;
  while (i != 0) {
    if (i < -w) {
      i--;
      i = i*(-1);
    } else {
      if (i > w) {
        i++;
        i = i*(-1);
      } else {
        i = 0;
      }
    }
    w++;
  }
}
```

Figure 5.8: AlternDivWidening
This is a program whose input variable `i` increases and flips the sign in each iteration. It does not terminate for all input values greater than 5 or smaller than −5.

Because of the flipping, none of the invariants $i > M$ or $i < -M$ (for some value of M) leads to a closed non-termination proof. The possible invariants contain either a lot of negated equalities or a disjunction. One which covers the most values is

$$\mathtt{i} < -5 \vee \mathtt{i} > 5.$$

Another and uglier one is

$$\mathtt{i} \neq -5 \wedge \mathtt{i} \neq -4 \wedge \ldots \wedge \mathtt{i} \neq 4 \wedge \mathtt{i} \neq 5$$

The first invariant could never be constructed using only the creation methods 1 to 4. By the creation method no 5, this invariant might be constructed.

5.2.2 Filtering of Invariant Candidates

In the invariant creation phase, a lot of invariant candidates are created that are not helpful in the search of a non-termination invariant. This is due to the fact that these methods are applied “blindly” without actually examining the old invariant candidate. Therefore we filter out those candidates which are obviously useless for various reasons.

```
foo(int i) {
  if (i > 20) {
    while (i > 10) {
      i++;
    }
  }
}
```

Figure 5.9: InitNotClosed
This program is an example for that the loop might not be reached although the loop condition is fullfilled, for example if `i` = 15.

1. Equivalence to *false* A newly created invariant candidate can be equivalent to the formula *false*. Because the first property of non-termination invariants is that the invariant must hold before the loop execution, *false* is never a good choice.

Example 5.8. Examples for invariant candidates which are equivalent to *false* are the following.

$$\begin{array}{lcllcl} inv_A & \equiv & \mathit{true} \land \mathtt{i} > 5 \land \mathtt{i} < 6 & inv_B & \equiv & \mathit{true} \land \mathtt{i} < 0 \land \mathtt{i} > 0 \\ inv_C & \equiv & \mathit{true} \land \mathtt{i} * \mathtt{i} < -3 & inv_D & \equiv & \mathit{true} \land \mathtt{i} = 0 \land \mathtt{i} \neq 0 \end{array}$$

with `i` : `int`.

2. Equivalence to other Invariants A fresh invariant candidate can be equivalent to a candidate that was already created and especially used in an earlier iteration. To avoid unnecessary calculations and thus save resources, we filter out those candidates. Because invariant candidates grow in complexity in each iteration it is reasonable to save the earliest and thus simplest version of the candidate, all later occurrences of equivalent candidates are dismissed.

Example 5.9.

$$\begin{array}{rcl} \mathit{true} \land \mathtt{i} > 5 \land \mathtt{i} > 3 & \text{is equivalent to} & \mathit{true} \land \mathtt{i} > 5. \\ \mathit{true} \land \mathtt{i} = 10 \land \mathtt{i} \neq 0 & \text{is equivalent to} & \mathit{true} \land \mathtt{i} = 10. \end{array}$$

3. Impossible Closure of the Init-branch The application of the invariant rule makes the proof branch threefold. As described in Section 4.2.2, the first branch proves that the invariant holds when the loop is reached in the execution of the program.

In the refinement process, invariant candidates might be created that do not hold in the beginning of the loop. That means they describe a set of states that is never reached right before the loop is entered. Once we have created an invariant candidate which prevents the first branch from closing, it does not make sense to make any more refinement of it, because we only refine candidates by adding new conditions conjunctively, which reduces the set of program states even more.

Example 5.10 (InitNotClosed)**.** The program InitNotClosed is shown in Figure 5.9. It has the loop condition $\mathtt{i} > 10$, but the loop is only reached if $\mathtt{i} > 20$. In the refinement process, the invariant candidate

$$\mathit{true} \land \mathtt{i} < 15$$

might be constructed. If the input variable i had a value less than 15, then the candidate would not hold before the loop because the execution of the program would already assume that $\mathtt{i} > 20$. The branch in the proof would start with the sequent

$$\mathtt{i} > 20 \Rightarrow \mathtt{i} < 15$$

and would not be closable. Under the condition that i fulfills the invariant candidate, the loop would not be reached and therefore not be carried out and therefore terminate trivially. This invariant candidate is filtered out by the filter method no 3.

4. Complexity of the Formula In contrast to the other three criteria for dismissing an invariant candidate, this one exists for performance reasons only. In the application of the algorithm the size of the invariant candidates grows fast. At some point the candidates become too big to be handled by the theorem prover in a reasonable way. Therefore we introduce filters for invariant candidates that became to big. The filter can either base its decision on the number of operators in a formula or on the depth of the formula (if considered as a tree).

5.2.3 Scoring of Invariant Candidates

After the invariants are created and filtered, they are assigned a score. A score is a real number between 0 and 1 which indicates the probable usefulness of the invariant candidate for proving the non-termination of the program. There are several criteria by which the candidates are judged. The overall score of an invariant candidate is the weighted average of the scores of each criterion, where the weights are assigned to the criteria to adjust the influence of the different criteria on the scoring.

In this section, we will list and describe the criteria that our algorithm applied. We also state the reason why we considered them to be useful heuristics. In the experiments we made, we found out that some of them are less important than initially presumed. We will discuss the results of the experiments in Chapter 7.

1. Complexity A goal in the design of the algorithm (Section 2.4) was to describe the set of critical inputs as general as possible. For this reason it makes sense to prefer simpler invariant candidates to complex ones to make sure that more general descriptions of the set of critical states are found earlier than others. Therefore the algorithm judges the invariants by their complexity. This can be done by examining either the number of operators in a formula or the depth (when considering a formula as a tree of operators). A simple invariant candidate is then assigned a smaller number than complex candidates.

Example 5.11 (Scores for Complexity)**.** In this example we calculate the score according to how we implemented it in our software. Note that the way we calculated the scores here is just one way to calculate it. It is possible to calculate the score differently, as long as a less complex formula is assigned a smaller score. In this example calculation the score grows linearly with the number of operators or the depth. Another way to calculate the score might be some logarithmic or polynomial approach.

Given the maximum number of operators $\mathrm{max}_{\mathrm{op}}$, the maximum depth $\mathrm{max}_{\mathrm{d}}$, the actual number of operators $\mathrm{num}_{\mathrm{op}}$ and the actual depth of the invariant $\mathrm{num}_{\mathrm{d}}$ the scores s_{op} concerning the number of operators and s_{d} concerning the number of s_{d} is calculated as follows.

$$s_{\mathrm{op}}(inv) = \tfrac{\mathrm{num}_{\mathrm{op}}(inv)}{\mathrm{max}_{\mathrm{op}}} \quad s_{\mathrm{d}}(inv) = \tfrac{\mathrm{num}_{\mathrm{d}}(inv)}{\mathrm{max}_{\mathrm{d}}}$$

We calculate the scores for two example invariant candidates.

$$inv_1 \equiv true \wedge \mathtt{i} > 5 \quad \text{and} \quad inv_2 \equiv true \wedge \mathtt{i} < 20 \wedge \mathtt{i} * 4 = 20$$

Assuming that the maximum number of operators of an invariant candidate is $\mathrm{max}_{\mathrm{op}} = 20$ and the maximum depth is $\mathrm{max}_{\mathrm{d}} = 10$. The number of operators of inv_1 is $\mathrm{num}_{\mathrm{op}}(inv_1) = 5$ and of inv_2 is $\mathrm{num}_{\mathrm{op}}(inv_2) = 11$. The depth of inv_1 is $\mathrm{num}_{\mathrm{d}}(inv_1) = 3$ and of inv_2 is $\mathrm{num}_{\mathrm{d}}(inv_2) = 5$.

The scores concerning the number of operators s_{op} are then the following.

$$\begin{aligned} s_{\mathrm{op}}(inv_1) &= 0.25 \quad & s_{\mathrm{op}}(inv_2) &= 0.55 \\ s_{\mathrm{d}}(inv_1) &= 0.33 \quad & s_{\mathrm{d}}(inv_2) &= 0.5 \end{aligned}$$

In both ways to calculate the score, inv_1 has a lower score as inv_2 and is therefore preferred to it in the queue.

2. Invariant Metavariables The creation methods of introduction of metavariables (method no 4 in Section 5.2.1) is a strong tool (and sometimes the only effective one) to find the desired invariant. The problem with invariant metavariables is that in cases where they do *not* lead to a closed proof, they tend to lead to even bigger open proofs. That means they feed the explosion of invariant candidates by leading to proofs with a lot of open goals. It is reasonable to prefer invariant candidates that do not contain invariant metavariables to those who do contain them in order to keep the number of newly created candidates as low as possible.

3. Initial-state Metavariables A non-termination proof of a program with at least one input variable contains at least one initial-state metavariable, because we quantify existentially over the input variables of the program. The metavariables stand for the value of the program variable at the beginning of the program. The variables occur of course in open goals and thus in new invariant candidates as well.

There are programs, which exclusively consist of a loop, which means that the `while` statement is the first statement of the program. We call those programs *loop-dominated* programs for this section.

Invariants in non-termination proofs describe the set of program states that is never left during the execution of the loop. In loop-dominated programs there are no manipulations of the program variables before the loop. Therefore, the conditions which the invariants describe apply as much to the initial value of the program variables as on the values of the program variables in an arbitrary loop iteration.

For example, in a non-termination proof of a program which has one input variable `i`, the metavariable I is introduced. The algorithm might find out, that the program does not terminate for values that are greater than 5, thus the invariant candidate has the form

$$\mathit{inv}_{\text{new}} \equiv \mathit{inv}_{\text{old}} \wedge \mathtt{i} > 5.$$

If this candidate is not sufficient for closing the proof, another iteration of the algorithm is carried out. One possible new invariant candidate might be

$$\mathit{inv}_{\text{new}} \equiv \mathit{inv}_{\text{old}} \wedge \mathtt{i} > 5 \wedge I > 5$$

This is a new candidate, but it does not really contain new information, because when the value of `i` must not be less than 5, this does also apply to the variable right in the beginning. Because I represents the value of `i` at the start of the program, it is unnecessary to include this information here.

For this reason the algorithm prefers candidates that do not contain initial-state metavariables to those who do contain them. This is in particular reasonable for loop-dominated programs. The more statements precede the loop, the more might the requirements for the initial state differ from the requirement for states which are reached in the loop. In these cases it makes sense not to reject initial-state metavariables in the invariants. Unfortunately, we could not find an example where it is actually necessary, because the programs of our sample database were mostly loop-dominant[8]. Besides that we suspect that there are cases where such an invariant fragment is not actually necessary but helps the theorem prover to create a shorter proof.

We implemented this scoring method although it is only useful for the particular group of loop-dominant programs, because most of our programs were loop-dominant. In addition, we assume, that the extraction of the loop from a program is a reasonable preprocessing step to yield better performance (see Section 7.7). These extracted loops then form loop-dominant programs.

Example 5.12 (Initial-state and Invariant Metavariables)**.** There are several ways to implement the preceding two scoring methods. Assume that we have implementations which simply output the score 1.0 if the invariant candidates contains the respective type of metavariable and 0.0 if not. Have a look at the following invariants

$$\begin{gathered}
\mathit{inv}_0 = \mathit{true} \wedge \mathtt{i} > 5 \wedge \mathtt{k} < 2 \\
\mathit{inv}_1 = \mathit{true} \wedge \mathtt{i} > 5 \wedge I < 2 \\
\mathit{inv}_2 = \mathit{true} \wedge \mathtt{i} > 5 \wedge M > 42 \\
\mathit{inv}_3 = \mathit{true} \wedge \mathtt{i} > 5 \wedge I < 23 \wedge M > 0
\end{gathered}$$

[8] Or they were nearly loop-dominant, which means there were only few (trivial) program statements before the loop.

where I is the initial-state metavariable which was introduced for the existentially quantified variable `i` and M is a newly introduced invariant metavariable. Let $s_{\mathrm{art}}(inv)$ be the score calculated by scoring method no 2 and $s_{\mathrm{nat}}(inv)$ the one for scoring method no 3. The invariant candidates yield the following scores:

$$\begin{aligned} s_{\mathrm{art}}(inv_0) &= s_{\mathrm{art}}(inv_1) = 0.0 \\ s_{\mathrm{art}}(inv_2) &= s_{\mathrm{art}}(inv_3) = 1.0 \\ s_{\mathrm{nat}}(inv_0) &= s_{\mathrm{nat}}(inv_2) = 0.0 \\ s_{\mathrm{nat}}(inv_1) &= s_{\mathrm{nat}}(inv_3) = 1.0 \end{aligned}$$

The algorithm calculates the total score by adding up all single scores. The single scores can be weighted. Assuming that the two scores in this example are equally weighted, we yield the overall scores (calculated as average with equal weights of both scoring methods).

$$\begin{gathered} s(inv_0) = 0.0 \\ s(inv_1) = s(inv_2) = 0.5 \\ s(inv_3) = 1.0 \end{gathered}$$

4. Multiple Occurrence of Formulae In an open proof, sometimes the same formulae occurs in several open goals. It is reasonable to prefer invariant candidates made from those formulae to others, because if the candidate makes the algorithm close branches, it will close several branches at the same time and not only one.

5. Reoccurring Formulae Formulae which occurred in open proofs in several iterations of the algorithm might be reasonable candidates for the next invariant, because they hint to situations where the non-termination proof repeatedly failed. Thus, we introduce by that a kind of history criterion, which prefers invariant candidates made from formulae that occurred often in open proofs to those which only occurred rarely.

6. Proof Size For this work, we define the size of an open proof and the size of an open goal as follows.

Definition 5.13 (Goal Size and Proof Size)**.** An open goal g has the form $\Gamma \Rightarrow \Delta$ where Γ is the set of formulae of the antecedent and Δ is the set of formulae of the succedent. The size of an open goal is defined as

$$|g| = |\Gamma| + |\Delta|.$$

The size of a closed goal is 0. The size of a an open proof p is the number n of open goals g_i for $i = 0, \ldots, n$. The size of a closed proof is 0.

We presume that the smaller an open proof is the closer it is to being closed. Therefore, it is a reasonable assumption that formulae which come from small open proofs are to be preferred to those that come from large open proofs. Thus we prefer invariant candidates from proofs that where nearly closed to those that come from proofs with lots of open goals.

5.3 Soundness and Completeness

In this section, we examine soundness and completeness as properties of algorithms. Soundness in this case means that, whenever the algorithm's result is "The input program does not terminate", then the input program actually does not terminate. The presented algorithm is sound if the underlying theorem prover is. The algorithm only outputs "The input program does not terminate" if the theorem prover could prove the non-termination with one of the invariants which was created in the algorithm. If a false positive[9] would occur, it would be due to the unsoundness of the theorem prover.

Completeness means that every non-terminating program could be identified by our algorithm. Our algorithm is of course not complete. Anyway, it would be surprising, because otherwise we would have solved the halting problem.

In conclusion we can say about the algorithm, that if the input program does not terminate, our algorithm either outputs "This program does not terminate." or "I do not know if this program does not terminate". If the input program actually terminates for all possible inputs, the algorithm outputs always "I do not know if this program does not terminate."

5.4 The Algorithm for Nested Loops

In principle, this algorithm can be applied to nested loops, too. There are basically two ways to deal with nested loops when using our algorithm: transformation into an unnested loop or examining each (inner) loop separately. We will describe both ways in this section.

5.4.1 Transformation into Unnested Loops

Every nested loop can be transformed into an unnested loop. This transformation is done by introducing an extra variable which indicates if the current loop iteration is one of the former inner loop or of the former outer loop. Figure 5.10 shows the transformation of a nested loop with one inner loop and one outer loop into a single unnested loop. By successive application of this transformation, nested loops can be transformed into one single unnested one. This transformation becomes more difficult if the nested loops are in conditional branches, but we assume that in principle a similar transformation is possible for arbitrary programs. For the examples which we examined, the transformation as shown in Figure 5.10 was sufficient.

5.4.2 Examination of Inner Loops Separately

This algorithm is not restricted to a fixed number of loops nested in each other, but for simplicity reasons, we only show examples containing only two loops, where one

[9] A false positive is a program that is announced to be non-terminating, but which is actually terminating for all possible input values.

```
boolean inner = false;
while (a || inner) {
  if (!inner) {
    p_a;
    inner = b;
  }
  if (inner) {
    p_b;
    inner = b;
  }
  if (!inner) {
    p_c;
  }
}
```

```
while (a) {
  p_a;
  while (b) {
    p_b;
  }
  p_c;
}
```

Figure 5.10: Transformation of a Nested Loop into a Single Unnested Loop
The left program shows the transformation of the nested loop in the right program. The statements `p_a;`, `p_b;` and `p_c;` stand for arbitrary (but not loop-containing) code fragments. The transformation is done by introduction of the additional variable `inner`.

is an inner loop of the other.

A program which has n loops $l_1, \ldots, l_n$, where l_n is the inner most loop and l_1 the outer most loop, has n different possibilities to not terminate, namely each single loop. Of course several of these loops can be non-terminating, but it is sufficient to find one of them.

Consider the example program which we show in Figure 5.11. The cause for non-termination here is the inner loop `while (j > 5)`, which does not terminate for $j > 5$. If there was not the inner loop, the outer loop would terminate.

In contrast, example NestedOuter is a program with a nested loop where the outer loop causes the non-termination while the inner loop always terminates.

In this approach we prove the non-termination of nested loops by examining each loop separately. Approaching a nested loop, it makes sense to first check the non-termination of the inner most loop, because it is less complex than to deal with the outer loops, and then go on to the more outer loops. The way to deal with each loop is the same, so we describe the approach for some arbitrary loop l_i for $i \in \{1, \ldots, n\}$ where n is the number of loops in the program. The analysis is done in these steps.

1. Extract the code of loop l_i including its inner loops from the program.
2. Form a formula stating the non-termination of this code fragment without quantifiers. This means a formula of the form $[p]\mathit{false}$ where no quantifiers precede the modality and p is the extracted code fragment.
3. Precede the formula with an update $v_j := a_j$ for all variables $v_j \in \mathcal{V}_p$ which

```
increase(int i) {
  int j;
  while (i < 10) {
    j = i;
    while (j > 5) {
      j++;
    }
    i++;
  }
}
```

Figure 5.11: WhileNestedOffset
This program contains a nested loop. The inner loop does not terminate if `j` > 5. The outer loop does not terminate because it will always eventually set `j` to a value greater than `5` and thus cause the inner loop to not terminate.

```
nestedOuter(int j) {
  while (j > 0) {
    int i = 0;
    while (i < 2) {
      j += i;
      i++;
    }
  }
}
```

Figure 5.12: NestedOuter
This program contains a nested loop. The outer loop does not terminate for all values for `j` which are greater than `0`. The inner loop always terminates.

- are assigned a value a_j before the outmost loop of the program and
- are not changed in any of the loops.

4. Precede the formula with an update $w_k := W_k$ for all variables in $w_k \in \mathcal{V}_p$ which do not fulfill the requirements in the preceding point. W_k is a fresh initial-state metavariable for each variable w_k.
5. The formula looks then like

$$\Rightarrow \{v_1 := a_1 \mid\mid \ldots \mid\mid v_p := a_p \mid\mid w_1 := W_1 \mid\mid \ldots \mid\mid w_q := W_q\}[\texttt{while}\ldots]\mathit{false}$$

6. Then we apply the algorithm as described in Section 5.1 chapter with this formula as initial formula.
7. In the branch of the proof which represents the preservation of the invariant in the execution of the loop body, inner loops l_j of l_i with $j < i$ will occur in the box modality. In this case a partial correctness proof of the inner loop is necessary in order to perform the iteration of the algorithm.
8. If the algorithm can prove the non-termination of the loop, it will output the invariant and constraints which describe the requirements for the variables which were assigned metavariables. By introduction of those variables we proved that

the loop l_i does not terminate for some program states as starting state. The problem is that we cannot ensure that these states are actually reached in one of the executions of l_i's outer loops. Therefore we additionally have to prove the reachability of these states in a separate proof.

This method to deal with nested loops has some weaknesses compared to the approach of translating the nested loops into a single unnested one. The weaknesses are the necessity of a partial correctness proof of the inner loops and the necessity of the additional reachability proof.

The advantage of this approach is that we can identify directly which of the loops causes the non-termination where the approach of transformation into a single loop does only output the critical inputs without an explanation which loop causes the problem. This feature is in particular useful in case this analysis is integrated into a development environment. In this case the tool could tell the developer directly which loop he has to debug.

Concerning the complexity, this approach has the advantage that the algorithm can stop as soon as it has proven the non-termination and reachability of one inner loop. The more inside the critical loop is the shorter the runtime of the algorithm is. The approach of translation into a single loop on the other hand examines in its proof always all loops at the same time, because it investigates the single unnested loop only.

In the KeY prover, which we use for our implementation, there is so far no procedure to generate partial correctness proofs of loops fully automatically. Therefore we could not solve the examples with nested loops automatically using this approach, because we only use the functionality of KeY which can be applied without human interaction. We did though apply the first approach of transformation into a single loop and will present the results in Chapter 7.

Chapter 6

Implementation of the Algorithm

We implemented the algorithm which we presented in the preceding chapter and ran a number of experiments on a database of non-terminating programs. In this chapter, we inform about the details of the implementation, the technology which we used, the challenges we faced during the development and how we managed them, how the different components work together and how the software is used by a user.

Note, that this implementation is highly experimental and shall be considered more as a proof of concept rather than as designed for actual deployment. For as far as we know, this is the first implementation of an automatic tool for detection of infinite loops.

6.1 Used Technology and Technical Requirements

We implemented the algorithm which we described in Chapter 5, except for the theorem prover. We used an existing theorem prover as back end, namely the KeY prover which we presented in Chapter 3.5. The KeY prover uses an external implementation of Cooper's algorithm as constraint solver to handle metavariables as described in Chapter 3.4.

Our implementation of the algorithm is written in Java 5. We used ANT scripts and the standard Javac compiler, version 1.5.0_06, for compilation. The software runs on every machine which provides an adequate Java virtual machine, and fulfills the requirements of the KeY prover. In addition, the program xvfb-run must be installed, which is a requirement to ensure the correct communication between the invariant generator and the KeY prover. We will talk about the communication in more detail in Section 6.4.

The version of KeY which we used in the development and experimental phase was taken from the CounterExamples branch version 67 to 76.

6.2 Design

We designed the software to make it as flexible as possible for experimenting with different components in the four phases (Section 5.2) of the algorithm. Figure 6.1

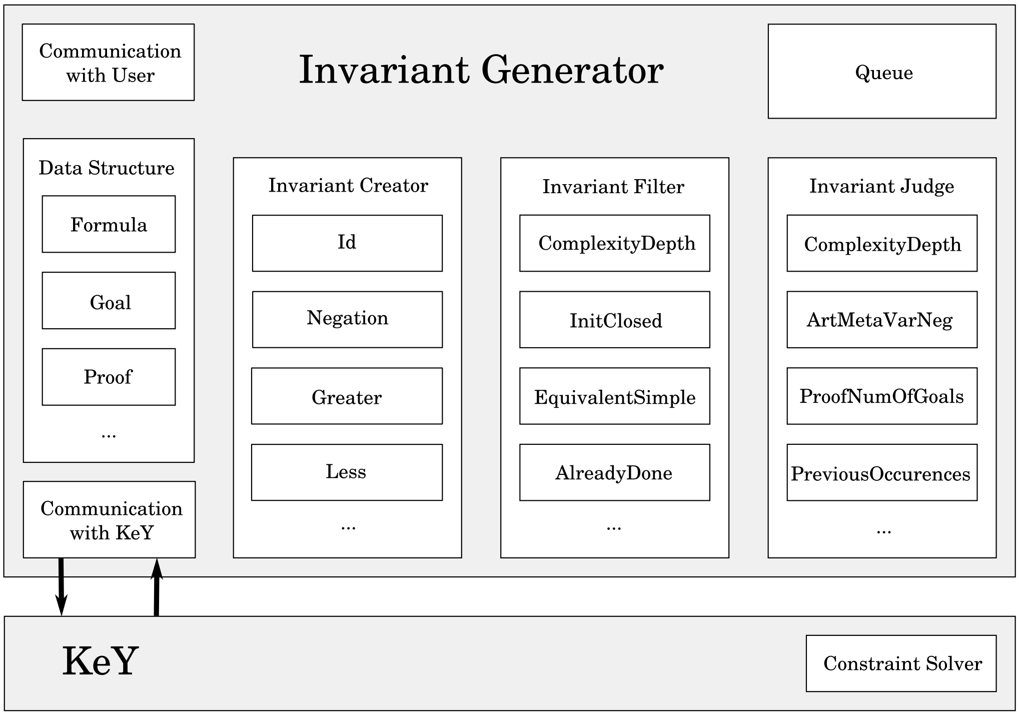


Figure 6.1: Components of the Software

This is a sketch of the components of our software. The invariant generator has four main components, invariant creator, invariant filter, invariant judge and the queue. Besides that there are components for the data structure and the communication with the user and the KeY prover. The KeY prover is an external component and contains the constraint solver as subcomponent.

shows a sketch of the design of the software.

Each phase of the algorithm has a set of modules, namely creation modules, filter modules, score modules and for the fourth phase, different implementations of queues. There is always an abstract class for each phase, of which the concrete implementations of the modules inherit. We will briefly introduce all modules here and describe their purpose. Note, that the modules not exactly match the described components in Chapter 5, because we gave ourself the freedom of experimenting with different approaches while designing a solid algorithm.

6.2.1 Creation of Invariant Candidates

All classes of the software which are responsible for the invariant creation inherit from the abstract class `CreationModule` and provide a method which takes a set of sequent formulae as input and outputs a set of invariant fragments. Most of them are implemented according to their description in 5.2.1.

- `IdModule` This module implements the creation method no 1, the adding of formulae of the succedent of an open goal.
- `NegationModule` This module works according to the description of creation method no 2, which is the adding of the negation of a formula of the succedent of an open goal.
- `GreaterModule` This module is the implementation of the creation method no 3, where equalities from the antecedent of an open goal are turned into a greater-than inequality.
- `LessModule` This module does the same as `GreaterModule` except that it creates a less-than inequality.
- `GreaterMeta` This module examines an equality of the antecedent of an open goal, takes the term of the left side of it and includes this term into a greater-than inequality with a fresh metavariable on the right side. This module is the rudimentary predecessor of the creation method no 3, the introduction of metavariables, which is described in Section 5.2.1. It has the drawback that in the theorem prover the terms in equalities are sorted in a fixed but arbitrary order and if we only consider the left side of an equality we might miss important terms which occur on the right.
- `LessMeta` This module does the same as the preceding one, except that it produces less-than inequalities instead of greater-than ones.
- `SmartGreaterMeta` This module is the mature version of `GreaterMeta`. It creates greater-than inequalities for each occurring term of an open goal. It implements exactly the creation method no 4.
- `SmartLessMeta` This module does the same as the preceding one, except that it creates less-than inequalities.
- `OrPair` This is the implementation of the creation method no 5, which is adding pairs of invariant fragments disjunctively.

6.2.2 Filtering of Invariant Candidates

All invariant filters are implemented as separate classes. They inherit from the class `FilterModule` and provide the method which takes an invariant candidate as input and outputs if this candidate is approved or not. The filter methods are described in the preceding chapter in Section 5.2.2.

- `ComplexityDepth` This module approves an invariant candidate if its depth (when considering the formula as a tree) is under a certain limit. This limit can be specified as a parameter by the user. This is suggested in filter method no 4.
- `ComplexityNumOfNodes` This module also examines the complexity of an invariant candidate but in contrast to the preceding module it dismisses a candidate if its number of operators exceeds a certain limit. This limit is also specified

by the user. The method is described in the preceding chapter as filter method no 4.

- `InitClosed` This module checks if the considered invariant candidate comes from a proof whose initial branch is not closed. This filter addresses the problem of impossible closure of the init-branch, which is described as filter method no 3 in Section 5.2.2 and as interpretation of an open goal in Section 4.2.2.
- `EquivalentSimple` This module provides a simple check if the new invariant candidate is equivalent to the one of the current iteration. The check is done if the fragment formula which is added to the old invariant candidate to gain the new one is already a conjunctive subformula of the old candidate.
- `AlreadyDone` This module checks if the invariant candidate was already tried in some iteration.

We suggested a combination of `EquivalentSimple` and `AlreadyDone` in the description of the algorithm in Section 5.2.2 as filter method no 2. An exact implementation of this idea would requires a reasoning step by the theorem prover to check the equivalences of formulae. Unfortunately, the invocation of the theorem prover is a very costly step, which we therefore avoided by only implementing these rudimentary versions of the filters. The same applies for the filter no 1, which checks if an invariant is equivalent to false. We did not implement it, because it would have made the invocation of the theorem prover necessary. The performance issues which cause the high costs of the invocation of theorem prover are discussed in Section 6.6 of this chapter.

6.2.3 Scoring of Invariant Candidates

We implemented various modules to judge the quality of a candidate as non-termination invariant. All modules inherit from the class `JudgeModule` and provide the method which takes an invariant candidate as input and output the score as a real number between 0.0 and 1.0.

We implemented all scoring methods mentioned in Section 5.2.3 and additionally a few more. The latter ones have a more experimental character and happened to be not as useful as expected, which is why we did not include them in the algorithm's description in Chapter 5.

- `ComplexityNumOfNodes` is a module which assigns a score to an invariant candidate according to the number of operators in it. The fewer operators an invariant candidate has, the smaller is the score and thus the more is this candidate preferred in the queue (scoring method no 1).
- The module `ComplexityDepth` has the same behavior as the preceding one, except that it examines the depth of the invariant candidate rather than the number of operators. This is also described as scoring method no 1.
- The modules `ArtMetaVarNeg` and `NatMetaVarNeg` return a high score if the invariant candidate contains an invariant metavariable or initial-state respectively.

Thus, these modules are used to not advance those candidates in the queue. We described this method as scoring method no 2 and 3.

- The modules `ArtMetaVarPos` and `NatMetaVarPos` do the opposite of what their counterparts in the preceding item do. They assign low scores in case the invariant contains invariant metavariables, respectively initial-state metavariables.
- `GoalSize` is a module which judges an invariant candidate by the size (measured in the number of formulae in the antecent and succedent) of the goal where it comes from. The smaller the goal is the smaller is the score which is assigned by this module. The usefulness of this module was a guess and so far we could not show that it actually helps in the decision which invariant to chose next.
- The modules `ProofNumOfGoals` and `ProofNumOfFormulae` examine the size of the proof from which the candidate originates. The smaller the proof is (measured in number of goals or number of formulae in the goals respectively) the smaller is the score. We described `ProofNumOfGoals` as scoring method no 6.
- The module `multipleOccurences` checks whether the formula from which the invariant candidate originates occurs multiple times in the open proof. The more often it occurs the smaller is the score. This is the implementation of scoring method no 4.
- The modules `numArtMetaVarPos`, `numNatMetaVarPos`, `numArtMetaVarNeg` and `numNatMetaVarNeg` work similar to their counterparts without `num` (see 3rd and 4th item in this list). The difference is that these modules here not only check if an invariant candidate contains a metavariable but checks how many metavariables occur in the candidate.
- `PreviousOccurrence` implements the idea of checking how often an invariant candidate occurred in previous iterations of the proof. The more often the candidate occurs, the smaller is the score. We described this method as scoring method no 5.
- The modules `numModVarsNeg` and `numModVarsPos` also have a more experimental character. They assign high respectively low scores if the invariant contains few respectively many variables which are changed in the loop body (those are the ones in the modifier file, see Section 6.3).
- `ContainsOr` is a model which assigns a high score if the invariant candidate contains a disjunction. This seemed to be a useful idea because invariant candidates which were created by the creation module `OrPairs` tend to become quite complex and therefore are assigned a high score to not prefer them too much in the queue.

6.2.4 Other Components

The component diagram in Figure 6.1 gives an overview over all components of the software. We will not go into more detail about the actual implementation of any more classes, because we consider the ones we described above as the essential ones.

The just described modules are held in an invariant creator class, an invariant filter class and an invariant judge class respectively. There are different classes which implement invariant queues. The one that we used in the experiments was a priority queue which used the assigned scores to queue the invariant, but in principle one could use a normal FIFO or LIFO-queue as well (they would make the scoring obsolete then).

In addition, the software contains classes which capture the data structures of formulae, goals and proofs. There are several classes who handle the communication between our software and the theorem prover, for instance creating invariant files or parsing proof files. Another class deals with the communication with the user and one class glues the all parts together and contains the core algorithm, which was sketched in Figure 5.2.

6.3 Preparations of the Input Programs

Fortunately, the application of our algorithm requires no preparation of the input program's code. The program is given to the software as a JAVA file containing a public and static method which contains the program in question. There are only two files, which the user has to create in addition to the source code file.

The first file is a file written in KEY syntax, which contains the formula, which states the non-termination of the input program. In a full integration of this software in a development tool, this file could be created automatically.

The second file is necessary for the application of the algorithm, because we need to apply the modified invariant rule invRuleMod (Figure 3.10). It contains the list of variables, which are actually manipulated in the while loop[1]. The information is necessary to derive the reduced context for the invariant rule. See Section 3.3.2 for the presentation of the rule. We have to use rule invRuleMod here instead of the unmodified version of the rule because we work with incremental closure of constraints. See Section 3.4 for details. We call this file *modifier file*. The creation of this file could be done automatically because all information which are necessary for it can be retrieved from the program's code by static analysis.

6.4 Interaction with the Theorem Prover

In this setup the invariant generator is the front end of the software and the KEY prover is the back end of it. As we described in Section 5.1, the invariant generator communicates with the underlying theorem prover in several steps of the algorithm. The invariant generator hands over a formula in the 2nd step and invariant candidates in the 4th step of the algorithm. The theorem prover has to communicate the result of the proof (attempt) to the invariant generator after step 6 of the algorithm.

In our implementation, the communication is done via files. In every iteration of the algorithm, the invariant generator is invoked with these files as input: the KEY

[1] Those variables are complementary to the variables, which might be read in the loop, but never get assigned a new value except outside the loop.

file and an invariant file, which contains the invariant candidate which the prover is supposed to use in the application of the invariant rule and the information of the modifier file.

After KeY has constructed a (possibly failed) proof, it dumps a proof outline into a file. The proof outline contains the nodes of the proof where the proof branches and the leafs including the formulae of the open goals.

The usage of files is of course not the most performant way to implement the communication. So far, KeY does not provide any other way of interaction with external software. KeY was originally designed as an interactive theorem prover. This means that it has a user interface, which was so far the only way to insert an invariant for the invariant rule. To enable KeY to read invariants from files, modifications to the KeY systems had to be made[2].

The fact, that KeY was designed as interactive prover means that it comes with a graphical user interface. To our knowledge, the interface cannot be deactivated so far. Because the invariant generator invokes KeY on the operating system level and does not support a graphical interface, we had to capture the graphical output of KeY. We used a virtual frame buffer xvfb-run, which runs KeY but absorbs its output. This is a sufficient solution for this problem because KeY actually has an automatic mode, which means that it can be invoked with a proof obligation, constructs the proof and dumps the result. The graphical output is the only thing necessary to be taken care of.

6.5 User Interaction

The algorithm detects the non-termination of a program fully automatically (or not at all). Thus, the only user interaction that is necessary is the startup of the software. The software is invoked on the command line with the file name as only mandatory option (for everything else, reasonable defaults are provided). In addition, the software provides a number of options which the user can choose from, for instance the maximum number of iterations and which of modules for creation, filtering and scoring shall be used. Furthermore, options for particular modules can be specified as well.

6.6 Issues during the Development

During the implementation and experimental phase we encountered several problems. Some are due to the technical environment, some are inherent problems originating from the theoretical background. We describe the most severe technical ones here and discuss the others in the evaluation of our experiments in Chapter 7.

[2] We thank Philipp Rümmer for the implementation of this and some other features of KeY which were necessary for the implementation of our software.

Long Startup Time of KeY Following the description of the algorithm, our implementation invokes the theorem prover in each iteration. With each new invocation the theorem prover has to build up its logic and read in all necessary rules and heuristics. This is a significant amount of data to be processed before the actual proof can be constructed. Unfortunately, the startup time of the KeY prover is about 30 seconds[3]. This increases the overall time which the software consumes dramatically.

This problem would not occur, if the theorem prover had some kind of server mode. In such a scenario, the prover could be started once in the beginning and then be used by the front end to receive and process proof requests. In this mode at the startup, the prover loads all its rules, heuristics and builds up the logic suitable for the program. It does all these action only once in the beginning. Then the front end software could send a proof request in each iteration of the algorithm. The theorem prover in server mode could process the request and return the result without loading the rules and heuristics over and over again. This would save the long startup time in each iteration.

Unfortunately, KeY does not get such a mode, because it is not designed for an application like ours. Because our implementation is basically a proof of concept, we came to an arrangement with it and used it in this suboptimal way.

In the algorithm, the theorem prover can not only be used to construct the non-termination proofs, but also to make the checks which are necessary for the filter no 1 to 3 (Section 5.2.2). Because we already loose so much time in each iteration of the algorithm, we did not use KeY for those checks, but implemented rather rudimentary versions of it (Section 6.2.2).

Because of this performance issue, the runtime of our software is extremely long. We therefore had to limit the number of iterations to 50 to get reasonably many results with the resources which we were provided. An average run of the algorithm which really performs the 50 iterations takes about 60 minutes, of which about half an hour is consumed by KeY's startup time.

[3] Measured on the Pentium M machine with 1.50GHz and 1 GB RAM.

Chapter 7

Experiments

We applied our software, which we presented in Chapter 6, on a set of example programs. We ran several experiments with different heuristics and will discuss the results in this chapter.

7.1 Sample Database

As part of this work, we looked for non-terminating programs to test the performance of our software. This search for example programs turned out to be harder than expected, because there are nearly no sources for non-terminating programs. Obviously, people do not publish non-terminating software. If they do, they do not annotate it with this information, because probably they do not know about the non-termination yet.

For term-rewriting systems, there is an annual competition[1], in which different research groups compete with their automatic termination analysis tools. Over the years a database full of terminating and non-terminating term-rewriting systems was built up. Because there is no such competition for imperative programs (even not a competition with the goal of proving only the *termination* of programs), there is no public standard database available for non-terminating imperative programs which we could have used for our work.

To our knowledge, we wrote the first tool which automatically analyzes imperative programs for non-termination. All other projects which examine imperative programs exclusively look for the termination of them. See the introductory Chapter 1 for information about these projects. Therefore we had to build our own sample database of non-terminating programs as a starting point.

We collected 55 example programs from various sources. Among these sources were literature about programming, websites about common programming errors, bugtracker of open source software projects and personal communication. Unfortunately none of these sources was very fertile, which lead to the situation that we had to program most of the examples by ourself. We created the examples from our

[1] `http://www.lri.fr/~marche/termination-competition/`

own experience in software development and let us inspire by the non-terminating examples of term-rewriting systems of their termination competition.

We refer to our set of programs as WHILE DB and publish it on our website[2]. Among the 55 examples there is one which is actually terminating (WHILEDECR) and one where it is still unknown if there are inputs for which it does not terminate (COLLATZ). We proved the non-termination of all examples manually using KEY, except for WHILEDECR and COLLATZ. In these proofs we used the most general invariant which we could derive from the program code using our intuition and experience.

7.2 Setup for the Experiments

We ran a number of experiments with different heuristics and settings. As mentioned in Chapter 6, the KEY version which we used, was taken from the COUNTER EXAMPLES branch. We used version 67 to 76 of this branch.

For the experiments we used two GNU/Linux machines. One was a Ubuntu with kernel 2.6.17 on a Pentium M machine with 1.50GHz and 1 GB RAM and the other one was a Gentoo with kernel 2.6.14 on a Pentium 4 Single Core machine with 2.60GHz per CPU and 1.5 GB RAM. Most experiments were carried out on the latter machine. The used JAVA virtual machine was of version 1.5.0_04.

Originally, we planned to run a larger number of experiments using a cluster of computers. Due to certain technical issues this was not possible. See Section 7.5 for further explanation.

7.3 Overview over the Results

In this section, we will give an overview over the results of our experiments. We will compare experiments with different settings and different examples. Sections 7.4 and 7.6 then examine a selection of particular examples and discuss their results.

Number of solved Examples We are happy to announce that the results of our experiments are very good. Of the experiments which were carried out, 7 runs could be completed, which is a total number of 385 calls. Our software could find the non-termination of 41 of the 55 of the examples in total (75%), and at most 37 (67%) in one run. The tables in Figures 7.1 and 7.2 show which experiment could solve which example. Of the 385 experiments, 242 were successful, 14 terminated because the algorithm ran out of invariant candidates and 129 reached the maximum number of iterations.

Number of necessary Iterations In our experiments we set the upper limit for the number of iterations to 50. The reason are performance issues which we discussed in Section 6.6 and will analyze further in Section 7.5. We calculated the

[2] http://academia.helgavelroyen.de/

Run Name	Malmoe	Flensburg	Eupen	Helsinki	Stockholm	Kiruna	Paris	Total
AlternatingIncr	✓ 2	✓ 2	✓ 12	✓ 2	✓ 4	✓ 4	✓ 2	✓
AlternDiv	✓ 4	✓ 6	✗ -	✓ 4	✓ 4	✓ 6	✓ 4	✓
AlternDivWide	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
AlternDivWidening	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
AlternKonv	✓ 24	✗ -	✓ 29	✓ 28	✓ 8	✓ 18	✓ 40	✓
Collatz	✗ 19	✗ 19	✗ -	✗ -	✗ -	✗ -	✗ -	✗
ComplInterv	✗ 4	✗ 4	✓ 31	✗ 9	✓ 15	✓ 15	✓ 17	✓
ComplInterv2	✗ -	✓ 4	✓ 2	✗ -	✓ 2	✓ 4	✓ 11	✓
ComplInterv3	✓ 6	✓ 4	✓ 2	✓ 5	✓ 2	✓ 4	✓ 5	✓
ComplxStruc	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
ConvLower	✓ 25	✓ 6	✓ 3	✗ -	✓ 5	✓ 3	✓ 4	✓
Cousot	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓
DoubleNeg	✗ 2	✗ 3	✗ -	✗ -	✗ -	✗ -	✗ -	✗
Even	✓ 5	✓ 5	✓ 2	✓ 5	✓ 2	✓ 5	✓ 5	✓
Ex01	✓ 2	✓ 2	✓ 2	✓ 2	✓ 2	✓ 2	✓ 2	✓
Ex02	✓ 25	✓ 6	✓ 3	✗ -	✓ 3	✓ 3	✓ 4	✓
Ex03	✓ 3	✓ 5	✓ 2	✗ -	✓ 2	✓ 2	✓ 3	✓
Ex04	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓
Ex05	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓
Ex06	✓ 4	✓ 4	✓ 4	✓ 4	✓ 4	✓ 4	✓ 6	✓
Ex07	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓
Ex08	✓ 2	✓ 2	✓ 3	✓ 2	✓ 3	✓ 3	✓ 2	✓
Ex09Half	✗ 9	✗ 9	✗ -	✗ 50	✗ -	✗ -	✗ -	✗
Factorial	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
Fib	✓ 21	✓ 18	✓ 40	✓ 6	✓ 39	✗ -	✓ 8	✓
Flip	✓ 24	✓ 17	✓ 6	✓ 11	✓ 7	✓ 11	✓ 11	✓
Flip2	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
Gauss	✓ 3	✓ 3	✓ 2	✓ 3	✓ 2	✓ 3	✓ 3	✓
Gcd	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
Lcm	✓ 2	✓ 2	✓ 8	✓ 2	✓ 6	✓ 2	✓ 2	✓
Marbie1	✓ 2	✓ 2	✓ 3	✓ 2	✓ 3	✓ 3	✓ 2	✓
Marbie2	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓
Middle	✓ 2	✓ 2	✓ 31	✓ 2	✓ 13	✓ 2	✓ 2	✓

Figure 7.1: Results of all Runs on the Examples - Part 1/2

This table shows which run could solve which example. The symbol ✓ indicates that the example was solved and the number which follows is the number of iterations which was necessary. The symbol ✗ indicates that the algorithm could not solve the example. If it is followed by a number, it failed because it ran out of invariant candidates. If it is followed by "-", then it reached the maximum number of iteration, which was in all runs set to 50.

Run Name	Malmoe	Flensburg	Eupen	Helsinki	Stockholm	Kiruna	Paris	Total
MirrorInterv	✗ -	✗ -	✗ -	✗ -	✗ -	✓ 2	✗ -	✗
MirrorIntervSim	✓ 5	✓ 11	✓ 5	✓ 9	✓ 5	✓ 8	✓ 9	✓
ModuloLower	✓ 6	✓ 20	✗ -	✓ 7	✗ -	✗ -	✓ 17	✓
ModuloUp	✗ 9	✗ 9	✗ -	✗ -	✗ -	✗ -	✗ -	✓*
Narrowing	✓ 13	✗ -	✗ -	✓ 21	✗ -	✗ -	✗ -	✓
NarrowKonv	✓ 7	✓ 9	✗ -	✓ 7	✗ -	✗ -	✗ -	✓
Plait	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
Sunset	✗ -	✓ 48	✓ 5	✗ -	✓ 5	✓ 5	✓ 8	✓
TrueDiv	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓
TwoFloatInterv	✗ -	✗ -	✓ 4	✗ -	✓ 4	✓ 4	✓ 14	✓
UpAndDown	✓ 4	✓ 4	✓ 4	✓ 4	✓ 4	✓ 4	✓ 6	✓
UpAndDownIneq	✓ 4	✓ 4	✓ 4	✓ 4	✓ 4	✓ 4	✓ 7	✓
WhileBreak	✓ 2	✓ 2	✓ 3	✓ 2	✓ 3	✓ 2	✗ 2	✓
WhileDecr	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
WhileIncr	✓ 2	✓ 2	✓ 3	✓ 2	✓ 3	✓ 3	✓ 2	✓
WhileIncrPart	✓ 12	✓ 6	✓ 3	✓ 19	✓ 3	✓ 3	✓ 4	✓
WhileNested	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✓**
WhileNestedOffset	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✓**
WhilePart	✓ 25	✓ 6	✓ 3	✗ -	✓ 3	✓ 3	✓ 4	✓
WhileSingle	✗ -	✓ 5	✓ 2	✗ -	✓ 2	✓ 2	✓ 3	✓
WhileSum	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗ -	✗
WhileTrue	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✓ 1	✗
Number of ✓	30	35	35	34	36	35	36	41
Number of ✗ -	23	15	20	16	19	19	18	-
Number of ✗ n	2	5	0	5	0	1	1	-

Figure 7.2: Results of all Runs on the Examples - Part 2/2

See caption of preceding Figure 7.1.

*: The example ModuloUp could be solved in another run, which was done with a KeY version which had an improved modulo handling. This very run was performed on this example exclusively and could solve it in 2 iterations.

**: The examples WhileNested and WhileNestedOffset were solved after transformation into an unnested loop, see Section 5.4.

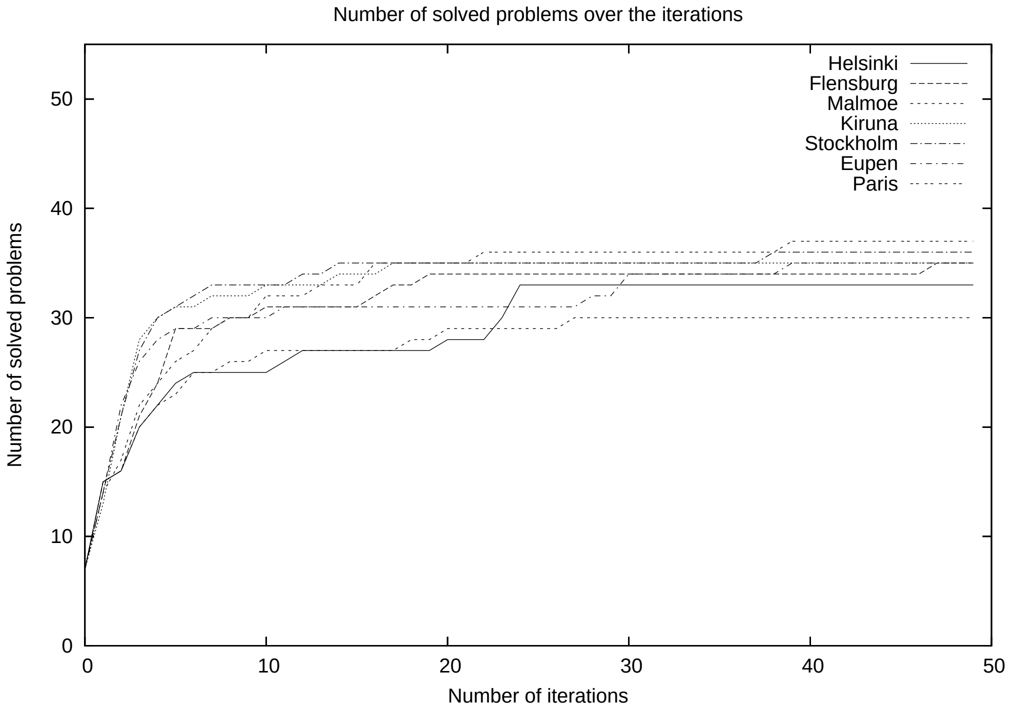


Figure 7.3: Performance of Different Runs as Graph.
This graph shows the performance of the different runs over the number of iterations. The x-axis describes the number of iterations and the y-axis shows the number of examples that were solved.

average number of iterations until the software terminated in each run to compare the performance of different runs. The number lies between 10.08 and 15.16 iterations, which includes the experiments which reached the maximum number of iterations. Considering only the successful experiments, an average number of iterations between 1.95 and 3.55 was the result. The diagram in Figure 7.3 shows after how many iterations how many problems were solved comparing the different runs. In this diagram we can see that most of the examples that were solved, were actually solved in the first 10 iterations.

Heuristics In the different runs of the experiments, we varied the heuristics, in particular which creation methods and scoring methods were used. Figure 7.5 shows which run used which creation method and Figure 7.6 shows which scoring methods were used and how they were weighted. All these runs were performed with a maximum number of iterations of 50, a maximum number of operators of the invariant candidates of 50 and the maximum depth of candidates was set to 15. All runs used these filter methods: `alreadyDone`, `complexityDepth`, `complexityNumOfNodes`, `equivalentSimple` and `initClosed`.

Of course the number of experiments is way too small to make an conclusive judgement of the performance of particular heuristics, but we draw some cautious conclusions in the following paragraphs. Figure 7.4 shows how many examples were solved by which runs and how their performance was measured in the average number of iterations and successful iterations. Because of the small number of runs, we could unfortunately not adjust the parameters first to get a canonical setting and then vary the setting to test different heuristics correctly. However, it is still possible to draw some conclusions.

The runs Malmoe, Stockholm and Kiruna were made to investigate the helpfulness of module method no 4, introduction of metavariables. Malmoe ran only with creation methods that do not use metavariables, Stockholm used only metavariables and the modules `IdModule` and `NegationModule` and Kiruna used all creation methods which the other two used. We chose the parameters for the scoring by common sense.

The result of this test was, that Malmoe could solve 30 examples, whereas Stockholm solved 36 ones. This improvement shows that the use of metavariables is really necessary for a significant number of examples. Kiruna then solved nearly as much examples as Stockholm did. Only one example which Stockholm could solve could not be solved by Kiruna. The problem here was that with more creation methods more invariant candidates are created and in some cases the maximum number of iterations was reached before the appropriate invariant candidate was tested. Interestingly, in Stockholm the average number of iterations until success (2.47) is significantly higher than that of Malmoe (1.67), whereas Kiruna lies right between those two (1.95). It seems to be that using metavariables is necessary for solving some of the examples, but in general it is reasonable to first try the invariant candidates without metavariables.

Examining the runs Kiruna and Paris, we can compare the usefulness of the creation method no 5, the creation of disjunctions. The total number of solved examples of Paris is slightly higher (+1) than the one of Kiruna. Unfortunately, the example AlternDivWidening, which we suspected to be solved by this run was not solved. Therefore, we can actually not attest that the creation method of adding disjunctions is useful. The only thing we can deduce from the comparison of the two runs is that the creation of disjunctive invariant candidates raises the average number of iterations in successful runs (in our case from 1.95 to 3.09). This is a consequence of the fact that the more creation methods we use the more invariant candidates are created and have to be tested until a correct one is found.

Settings of KeY The performance of our software is highly dependent on the performance of the underlying KeY prover. The different runs do not only vary in the different heuristics, we sometimes used different KeY versions, too. This is due to that in some runs problems occurred which could only be fixed by making changes to KeY. We are aware of the fact that this leads to a more vague comparability, but because of performance issues we did not have the resources to rerun the experiments

Run Name	Successful		Max. Iter.		Out of Inv.		Avg. num.	Avg. num.
	abs.	%	abs.	%	abs.	%	of iterations	sucsfl. only
Helsinki	34	61.82	16	29.09	5	9.09	11.44	3.02
Flensburg	35	63.64	15	27.27	5	9.09	10.35	1.93
Eupen	35	63.64	20	36.36	0	0.00	14.11	3.55
Malmoe	30	54.55	23	41.82	2	3.64	15.16	1.67
Stockholm	36	65.45	19	34.55	0	0.00	12.11	2.47
Kiruna	35	63.64	19	34.55	1	1.82	11.62	1.95
Paris	36	65.45	18	32.73	1	1.82	10.08	3.09
Total / Average	242	62.86	129	33.51	14	3.64	12.12	2.53

Figure 7.4: Performance of Different Runs expressed in Characteristics
The first two columns show how many of the 55 example could be solved (absolute number and percentage). The second two columns say the same for the examples where the maximum number of iterations was reached. The next two columns state the number and percentage of examples where the invariant generator ran out of invariant candidates. The last but one column shows the average number of iterations over all 55 examples. The very last column states the average number of iterations, if we leave out the examples where the maximum number of iterations was reached. The last row of the columns shows the sum of the absolute values and the averages of the percentages and averages.

Creation Method	Malmoe	Flensburg	Eupen	Helsinki	Stockholm	Kiruna	Paris
`Id`	✓	✓	✓	✓	✓	✓	✓
`Negation`	✓	✓	✓	✓	✓	✓	✓
`Greater`	✓	✓	✓	✓	✗	✓	✓
`Less`	✓	✓	✓	✓	✗	✓	✓
`LessMeta`	✗	✓	✗	✓	✗	✗	✗
`GreaterMeta`	✗	✓	✗	✓	✗	✗	✗
`SmartLessMeta`	✗	✗	✓	✗	✓	✓	✓
`SmartGreaterMeta`	✗	✗	✓	✗	✓	✓	✓
`OrPairs`	✗	✗	✗	✓	✗	✗	✓

Figure 7.5: Applied Creation Methods
This table shows which run applied which invariant creation method. For a description of the medhods see Sections 5.2.1 and 6.2.1.

Scoring Method	Malmoe	Flensburg	Eupen	Helsinki	Stockholm	Kiruna	Paris
`ComplexityNumOfNodes`	0.40	0.42	0.40	0.08	0.40	0.40	0.37
`ComplexityDepth`	0.40	0.33	0.40	0.08	0.40	0.40	0.37
`ArtMetaVarNeg`	-	-	-	-	-	-	0.07
`NatMetaVarNeg`	0.08	-	-	-	0.08	0.08	0.07
`ArtMetaVarPos`	-	-	0.04	0.08	-	-	-
`NatMetaVarPos`	-	-	0.04	0.08	-	-	-
`goalSize`	-	-	-	0.08	-	-	-
`proofNumOfGoals`	-	-	-	0.08	-	-	-
`proofNumOfFormulas`	-	-	-	0.08	-	-	-
`multipleOccurrences`	-	0.13	-	0.08	-	-	-
`numNatMetaVarNeg`	-	-	-	-	-	-	-
`numArtMetaVarNeg`	-	-	-	-	-	-	-
`numNatMetaVarPos`	-	-	-	0.08	-	-	-
`numNatMetaVarPos`	-	-	-	0.08	-	-	-
`previousOccurrence`	0.12	0.13	0.12	0.08	0.12	0.12	0.11
`numModVars`	-	-	-	0.08	-	-	-
`containsOr`	-	-	-	0.08	-	-	-

Figure 7.6: Applied Scoring Methods and their Weights

This table shows which run applied which scoring creation method and which weight. For a description of the methods see Sections 5.2.3 and 6.2.3

with different versions of KeY. Because our experiments were intended as a proof-of-concept rather than as an actual deployment, we did not prioritize the fine tuning of the heuristics anyway.

7.4 Discussion of Examples with Positive Result

Our software worked well on all examples which contain mostly linear operations. This is due to that the constraint solver could solve the constraints which come from those programs, because they were all linear. Example 7.1 is a simple linear example. Example 7.2 is a more sophisticated one, which was also solved by our software.

Example 7.1 (Ex02)**.** The example program Ex02 is shown in Figure 7.7. It does not terminate for all input values $\texttt{i} \geqslant 5$. The only possible invariants are

$$\texttt{i} \geqslant 5 \quad \text{or} \quad \texttt{i} = 5$$

or equivalent formulae. The number of iterations which our algorithm needed for a successful result lay between 4 and 26. The fastest run with 4 iterations was the

```
ex02(int i) {
  while (i > 0) {
    if (i != 5) {
      i--;
    }
  }
}
```

Figure 7.7: Ex02
This program does not terminate for all input values greater than or equal `5`. If such a value is given to the program it is decreased to `5` and then stays stationary.

```
alternKonv(int i) {
  while (i != 0) {
    if (i < 0) {
      i = i+2;
      if (i < 0) {
        i = i*(-1);
      }
    } else {
      i = i-2;
      if (i > 0) {
        i = i*(-1);
      }
    }
  }
}
```

Figure 7.8: AlternKonv
If this program is started with an odd value of the variable `i`, this value oscillates around `0` with a decreasing amplitude until it reaches `-1` or `1`. It then flips between those two and thus does not terminate. It terminates though for even input values.

run Kiruna. The found invariant was

$$\mathit{true} \wedge i > M$$

with the constraints

$$M < I \wedge M = 4 \wedge -1 < M.$$

This is equivalent to the more general of the two possible invariants, `i` $\geqslant 5$. Thus the invariant and the constraint says that the program does not terminate for all input values greater than or equal to 5.

Example 7.2 (AlternKonv)**.** Figure 7.8 shows the code of the program AlternKonv. If this program is started with an odd value for the variable `i`, then in the following iterations of the loop the value oscillates around zero with decreasing amplitude. When the value reaches -1 or 1, it goes on flipping between those two and thus never terminates. Figure 7.9 shows the behavior of the program for some input values. The termination behavior of this program is far from trivial and

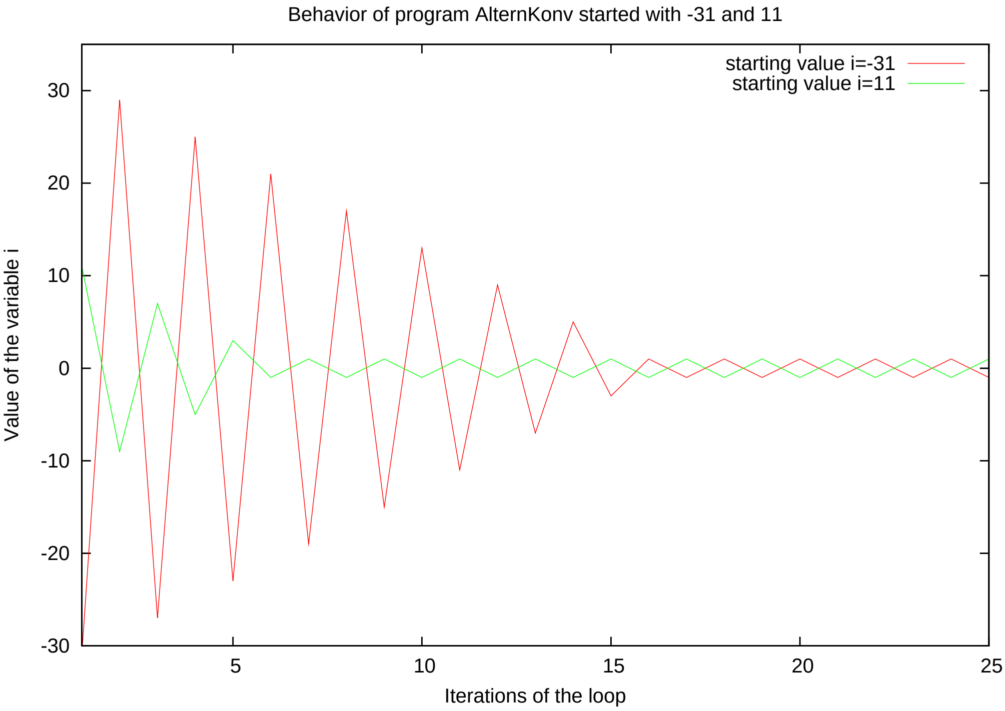


Figure 7.9: AlternKonv Value of `i` over the Iterations.
This plot shows the behavior of the variable `i` over the iterations of the loop. The two curves are computed using the starting values `i` = 11 and `i` = −31.

the program contains modulo and division operators which are not easy to handle. Nevertheless, our software could solve it.

There are several possible invariants, but the most general one is

$$i\%2 = 1$$

saying that `i` has to be odd. But also the following invariants are reasonable choices.

$$\begin{aligned}
inv_A &= i\%2 = 1 \wedge i > M_0 && \text{with } M_0 < -1\\
inv_B &= i\%2 = 1 \wedge i < M_1 && \text{with } M_1 > 1\\
inv_C &= i\%2 = 1 \wedge i > M_2 \wedge i > -M_2 && \text{with } M_2 > 1\\
inv_D &= i = -1 \vee i = 1
\end{aligned}$$

Our algorithm could solve this example in 8 to 40 iterations. The best run was Stockholm, which yielded the invariant

$$\mathit{true} \wedge i < M_0 \wedge i \neq 0 \wedge i > M_1$$

and the constraint

$$\begin{gathered}M_0 < 3 \wedge M_1 < -1 \wedge M_0 < 4 \wedge -4 < M_1 \wedge 1 < M_0 \\ \wedge -3 < M_1 \wedge M_1 < I \wedge I \neq 0 \wedge I < M_0.\end{gathered}$$

This is equivalent to the more human readable formula

$$i = -1 \vee i = 1.$$

This is not the most general invariant, but an important one, because if the programmer of this program would get this information and eliminate the problem for the values -1 and 1, the other critical inputs were not critical anymore and thus the whole problem would be solved.

7.5 Issues and their solutions

During the execution of the experiments, we encountered a number of issues. We will enumerate them here, discuss their consequences and how we found a workaround or solution for them. One issue was the long startup time of KeY, which we already mentioned in Section 6.6. Other issues were the following.

Instability of the Setup KeY is by now a well-grown and mature software project, but nevertheless it is still work in progress. Especially the CounterExamples branch, which we used for our experiments, was just recently added to KeY. The same applies for the integration of the constraint solver. The particular version of the constraint solver was an experimental choice rather than a fully investigated decision. The fact that thus all software components are in a beta stage lead to a number of instabilities in the setup.

The consequences were that a lot of experiments crashed due to these instabilities. Reasons were failure of communication between the parts, unexpectedly high memory and CPU demands and other problems with the technical environment. While we tried to address all these problems, unfortunately we could not solve all of them.

For example, we intended to run the experiments on a whole cluster of computers, but for some reason, we could not make the software run in a queuing system. The result were unreproducible errors, crashes and freezings, which we unfortunately could not identify, because on a single machine without a queuing system, the software worked fine. This is the main reason why the total number of successful runs of experiments is only 7.

Explosion of the Size of Open Proofs One problem was the fact that with growing invariant candidates the proof size grew dramatically. The consequences were that sometimes not all branches of the proof were constructed as far as actually possible, which means that sometimes complicated formulae remained in the open goals. Because it was not very promising to examine too complicated formulae, the component of our software which parses the data of the open proof dismisses formulae which are too complicated.

Explosion of the Number of Constraints A high number of open goals in a proof yields a high number of constraints for introduced metavariables. The higher the number of constraints is, the more complicated is the check for solvability of the constraints. This became a huge problem in later iterations of the algorithm on some of the examples. Since the problem of solvability of linear unification constraints is probably not decidable, we were aware of the fact that this issue is inherent of the problem and not of our implementation. The workaround which we chose was to limit the amount of time the constraint solver should try to solve the constraints. If the limit was exceeded, the proof was considered as not closed.

Explosion of the Number of Invariants A high number of open goals leads to a high number of invariant candidates. Thus, in each iteration, the invariant queue grows fast. Because every newly created invariant candidate is compared to all former ones to avoid double calculations, the number of comparisons grows with the size of the queue. Thus the long queue consumes a lot of memory and CPU resources. This became a problem in some examples. We decided to limit the number of elements of the queue to the maximum number of iterations. This influences the calculations of the scoring module `PreviousOccurrence` (Section 6.2.3), but so far was the best solution for this problem.

7.6 Discussion of Difficult Examples

There is a number of examples which could be solved only with a high number of iterations or not at all. In this section, we will have a look at the reasons for that and suggest ways to solve the problem.

Complex Control Flow or Many Variables Programs with a complex control flow structure and/or many variables could sometimes not be solved by our algorithm. The reasons are probably the limited number of iterations and restricted complexity of the invariant candidates. In a setting with higher resources we could let the software act more generously and so probably solve the problem. The following examples fall in this group of programs: ComplxStruc, MirrorInterv and Plait.

Modulo and Division Operations There is a number of programs which make use of the modulo and division operator (% and /). In some cases, our software had difficulties to cope with these programs. This is probably due to the fact that some versions of KeY had a poor modulo handling. The developers of KeY have improved this by now. We rerun one experiment on the program ModuloUp with the improved version of KeY and this time the problem could be solved. Another example of this category is the problem Ex09Half.

Other Nonlinear Problems Some programs have non-linear terms in their assignments or boolean expressions. Those programs were sometimes difficult to solve.

```
factorial(int j) {
  int i = 1;
  int fac = 1;
  while (fac != j) {
    fac = fac * i;
    i++;
  }
  return (i-1);
}
```

Figure 7.10: FACTORIAL
This program is supposed to calculate the number i for which i! = j holds. Unfortunately, the program chose the loop condition fac ≠ j too defensive. Therefore, if the value of the input variable j is not a factorial of any number, the program does not terminate.

This is probably due to the fact that the constraint solver cannot solve non-linear constraints and therefore KEY does not produce any of those, which of course restricts the power of the invariant refinement process. Programs with non-linear terms are for example: COMPLINTERV, DOUBLENEG and WHILESUM.

Programs with Non-convex Invariants There are a some programs, whose invariant cannot be described by a conjunction of conditions but by a disjunction. We came up with the invariant creation method no 5, which adds formulae disjunctively to the invariant candidate (see Section 5.2.1). Unfortunately, the experiments did not show any success for the affected examples so far. We assume that the problem must be addressed more general than just by adding two disjunct formulae. Fortunately, all of these programs seemed to be rather artificially constructed problems than anything that would realistically occur in a developers life. The affected programs are ALTERNDIVWIDE and ALTERNDIVWIDENING.

Nested Loops The algorithm as it is implemented in our software does not support nested loops. Therefore it is no suprise that the two examples WHILENESTED and WHILENESTEDOFFSET could not be solved by our software. However, it is possible to adapt the algorithm to nested loops. We described to ways to perform this transformation in Section 5.4. We applied the first one, which is the transformation into a single unnested loop, on the two examples. Our algorithm was able to prove the non-termination in two iterations each.

Non-trivial Successive Calculations There are a few problems, whose calculation in one iteration is non-trivially dependent on the results of the preceding iterations. By non-trivially we mean that it is more complex than just increasing a variable by one. An example for that is the program FACTORIAL, whose code is shown in Figure 7.10. In this example the value of fac is calculated by fac $*$ i. This problem requires an unusually complex invariant, which could not be created by our software. Other problems of this kind are FIB and EX09HALF.

Unknown problems Although our software did a good job, it still could not find a solution to the Collatz problem. Sad, but true.

7.7 Suggestions for Improvements

Because this is a proof-of-concept implementation, there are of course several things that could and need to be improved before this method could actually be used in a real world environment. From the experience of the experiments, we established several suggestions to improve the implementation.

Integration of Invariant Generator into Theorem Prover A solution to the problem of the long startup time of the theorem prover would be the integration of the invariant generation into the theorem prover, or as suggested in Section 6.6 a server mode of the theorem prover. A server mode means that the theorem prover is only started once in the beginning of the algorithm and then processes proof requests of the invariant generator in each iteration. This improvement would solve the issue with the highest impact on our software's performance.

Static Analysis to Retrieve Modifier Set As described in Section 6.3, the user has to provide a modifier file. The modifier file contains the list of variables which are manipulated in the the loop body. This information can of course be extracted from the source code as well. In a real time application this static analysis of the code should not be a problem and would remove an unnecessary task from the user's duties.

Loop Condition as Initial Invariant The third property of an non-termination invariant (see Section 2.5) is, that the invariant has to imply the loop condition. This means that at least the information of the loop condition has to be included in the invariant. It is therefore a reasonable approach not to take *true* as initial invariant candidate but the loop condition itself. This could save a number of iterations in our algorithm. Many of our examples would already be solved by only taking the loop condition as first invariant candidate, among those is the example which we introduced in the description of our algorithm in Example 4.1.

We did not include this feature in our software so far, because technically, the loop condition has first to be evaluated by KeY, then written into the invariant file and finally read from that file again. There is no feature in KeY so far, which provides this kind of functionality.

Initial Invariant produced by other Invariant Generation Tools Another idea to reduce the number of iterations is to use external invariant-generation tools to generate an initial invariant candidate. This formula would then be for sure an invariant to start with.

We made tests with an invariant generator which was developed by Weiß in [Wei07]. The generator uses techniques of abstract interpretation and is also based on KeY. However, the results so were not as good as expected. This is due to the characteristics of abstract interpretation, which is that the program is examined under all possible inputs. It is necessary to give the loop condition as precondition, because otherwise the program flow is too much generalized and the generated invariant was almost always *true*.

Because the number of tests was small and we tested only one invariant generator, we believe that there is still potential in this idea, although the tests have not been as successful as we thought in the first place.

Additional Information about the Program in Heuristics The search for the invariant is all about heuristics. The heuristics determine in what way the invariant candidate search space is traversed. In our implementation, we only used heuristics who look at the generated invariant candidates and the proof from which they originate.

It is a reasonable assumption, that it might be possible to equip the heuristics also with information about the program which can be retrieved by other means. Results of preceding tests of the program or static analyses of the program code might give hints for the choice of the invariant candidates. For example a program which has many variables, but not a very deeply nested control flow might be treated by different heuristics than programs which have only one variable, but a highly complicated control flow.

Improvement of Theorem Prover The performance of the invariant generator is of course highly dependent on the performance of the underlying theorem prover. This became already obvious when using different versions of KeY in the examples. None of the seven runs of our experiments was able to solve the example ModuloUp, but when we ran the experiment with a version of KeY, which had an improved handling of modulo and division operations, the problem was solved in 2 iterations.

This example shows how the quality of the theorem prover influences the quality of the invariant generation. Of course, all this depends also on the settings of the theorem prover. There are plenty of heuristics and strategies in a theorem prover, which itself need careful adjustment to yield the best performance.

Improvement of the Constraint Solver The same as for the theorem prover applies for the constraint solver (Section 3.4). The better the constraint solver is, the faster a proof can be closed and thus an invariant is found.

The performance of the constraint solver is highly inversely correlated to the complexity of the constraints. We discussed this trade-off already in Section 3.4. The drawbacks of expressive constraints like the linear unification constraints as used by KeY became obvious in our experiments, because in some iterations the constraint solver took several minutes to check the constraints and consumed all CPU power and available memory.

One might even like to be able to solve nonlinear constraints. This is even more complicated to solve, but was indeed a problem in some of our examples.

Other Ways of Introducing Metavariables So far, metavariables are only introduced into invariant candidates as the right side of an inequation. One could think of other ways to introduce them, for example in a modulo operator. The invariant candidate

$$i \;\%\; M = N$$

would have been a good guess for problems where the invariant describes a set of periodically distributed critical inputs. The example AlternKonv, which we described in Example 7.2 could have been solved more generally by this invariant with $M = 2$ and $N = 1$.

Loop Extraction In the initial phase of the algorithm (Section 5.1, steps 1 to 3), the non-termination proof of the program is constructed until the symbolic execution reaches the loop. In our implementation, this first part of the proof is done repeatedly in every iteration of the algorithm. For the programs in our sample database, this was never a hard task, because most programs do not have many statements before the loop statement. In general though, it is a good idea to save this proof stub after the initial phase of the algorithm and reuse it in every iteration.

It would be even better to extract the loop from the program with as much information of the context as possible and verify it separately. The advantage here is that if the construction proof stub needed human interaction, the whole proof would fail in our setup. If we could do the proof stub separately and if necessary with human interaction, we could apply our algorithm although there were manual steps necessary in the stub.

Intermediate Simplification of the Invariants The invariant candidates in our algorithm are constructed automatically. Automatic generation does not always find the most compact version of the formula. An idea is to use a theorem prover or similar application to simplify the candidates before they are put into the queue. This would make them smaller and thus reduce the complexity of the proofs. An example is the formula

$$\mathit{true} \land \mathtt{i} > 3 \land \mathtt{i} > 5,$$

which could be simplified to $\mathtt{i} > 3$.

7.8 Summarizing Evaluation of the Experiments

We created a database of non-terminating While programs and applied our algorithm on that. Although the number of examples was rather small, we think that a lot of common programming errors concerning non-termination are covered in this data base. We ran a number of experiments on this database to show the performance of our algorithm.

We stated a number of goals for non-termination analysis in Section 2.4, namely:

1. Identify non-terminating programs.

2. Identify the critical inputs. Those are the ones for which a program does not terminate.
3. Describe the set of critical inputs as general as possible.
4. Automate 1.-3. as much as possible.

In summary, we can say that the experiments yield promising results. Our software could solve 75% percent of the problems automatically, which is a big step concerning the first goal. For some of the more difficult problems there were some adjustments in the software and the prover necessary, but then even those could be solved automatically. We examined the results of the experiments on all programs in the database and identified possible solutions for the examples which could not be solved by our software.

The algorithm outputs a set of program inputs which result in an infinite loop; this meets the second goal. The simpler a non-termination invariant is, the more general is the description of the set of critical inputs. Because the heuristics for the search for invariants are designed to preferably look for simple invariant candidates, our algorithm finds those invariants first. This means our algorithm tends to find the most general description of the set of critical inputs, which meets the third goal. The program Ex02 is an example where the algorithm found the most general invariant and AlternKonv is one where a very specific one was found (Section 7.4).

Concerning the last goal, the full automation, we designed our algorithm to work without any human interaction. Either it detects the non-termination automatically or not at all. Therefore the last goal is fulfilled by the algorithm, too.

The experiments were accompanied by technical and mathematical problems, but in total the approach seems to be a good start. Especially programs which represented typical programming errors were solved easily by our software. This gives reason to be optimistic about the usefulness of the approach.

Chapter 8

Non-termination Analysis of HEAP Programs

In this chapter, we will transfer the work which we did on WHILE programs to another class of programs, namely HEAP programs. The HEAP language is an extension of the WHILE language with a richer type system and a heap. We define a suitable logic and calculus for this language and in Section 8.4 we perform the non-termination analysis of our algorithm on four example programs of the HEAP language.

8.1 HEAP Programs

HEAP programs are an extension of WHILE programs (Section 3.1). The essential difference is that HEAP programs can have complex data structures, which are captured in classes. Additionally, HEAP programs can dynamically allocate memory for their data structures. The memory which is used for this allocations is called heap memory, which gives the language its name.

The HEAP language can be considered as another (and larger) fragment of JAVA. It captures the basic concepts of object-orientation in modern programming languages. The following elements of JAVA are allowed in HEAP programs.

- All elements of the WHILE language, see Section 3.1.
- Arbitrary classes with attributes and methods (but no recursive methods).
- Inheritance of classes.

The inclusion of arbitrary classes and the existence of a heap allows complex data structures, in particular for instance arrays, linked lists or trees. An example for a HEAP program is shown in Figure 8.1.

```
public void insertInput(int n, int[] a, int cursor) {
  while (a[cursor] != 0) {
    cursor++;
    cursor = cursor % a.length;
  }
  a[cursor] = n;
  cursor++;
  cursor = cursor % a.length;
}
```

Figure 8.1: ChaosBuffer
The chaosbuffer is an array which stores incoming values at the next free space counting from the position of the `cursor` variable. It does not terminate if there are no free spaces in the array.

8.2 Heap Dynamic Logic

Heap Dynamic Logic is an extension of While Dynamic Logic (Section 3.2). The essential difference between the two logics is that Heap DL is able to deal with objects which While DL does not. Before we define syntax and semantics of Heap DL we explain how the concepts of object-orientation are handled in the logic.

8.2.1 Object-Orientation in Dynamic Logic

The way Heap DL deals with object-orientation is described in the following paragraphs. We used [BP06] as a guide for the design of the logic. Because we explain the object handling of dynamic logic already in detail here, we will keep the introduction of the syntax and the semantics in Sections 8.2.2 and 8.2.3 rather compact.

Classes and Types

Classes in object-oriented programs represent the types of variables. A value of a variable points to an instantiation of the class, namely an object. Classes are represented as types in Heap DL. There is an inheritance relation on the set of classes, which is represented by a subtype relation on the type set in the logic.

Arrays

The datastructure of an array is a particular class which uses the heap. Written in the syntax as it is shown in Figure 8.1, arrays do not comply to the definition of Heap programs which we gave in Section 8.1. We assume that arrays can be modelled in a class `Array` in a Heap program. This class has the attribute `length` and a method for the access of the array's elements.

Object Creation

During the execution of the program, objects are created. In our logic, all entities have to exist a priori[1]. This applies also to the objects in a program which are represented in the logic. To map the situations in a program's execution into the logic, we assume that all objects which could ever be created in the program exist beforehand. Therefore, we introduce an object repository Rep_C of a class C. The repository is a countably infinite set of all objects of class C. Each object has an index i with which we can retrieve the object from the repository, using the repository access function $\mathrm{get}_C : \mathtt{int} \to C$.

There is a counter next_C for each class C which keeps track on how many objects of class C are already created. Whenever a new object is created in the program the next "unused" object in the repository is taken and the counter is increased by one. The creation of a new object o of class C is thus represented in the logic by the update

$$o := \mathrm{get}_C(\mathrm{next}_C) \mid\mid \mathrm{next}_C := \mathrm{next}_C + 1.$$

In conclusion, all objects whose index is smaller than next_C are already created in the program and all others are not.

The function get_C represents a bijective mapping between the natural numbers and the repository of class C. Thus, two distinct objects of class C do not have the same index. Repositories of two distinct classes are disjoint; this holds also for subclasses. Thus, an object of a subclass B of C does not have an index in the repository of C; it has only one in the repository of B which is distinct from the one of C.

In object-oriented programs, variables have a static type, which is the one they are assigned at the declaration of the variable, for example in the program statement `Car o`. When initializing the variable with an object o, the type which is assigned to o by a statement like `o = new Porsche()` can be any subtype of the static class of the variable which points to o, in this case `Car`. This type is called dynamic type. In the execution of a program, dynamic type checks can be made. In JAVA for example they are made with the function instanceOf. We include a predicate instanceOf also into the logic, but we consider it as abbreviation of the formula

$$t \,\mathrm{instanceOf}\, C \equiv \exists n \bigvee_{\mathtt{Null} < \tau \leqslant \mathtt{C}} t = \mathrm{get}_\tau(n)$$

for $n : \mathtt{int}$.

Another abbreviation that we want to introduce is the predicate $\mathrm{created}_C$ which stands for the formula

$$\mathrm{created}_C(o) \equiv \exists n \; (\mathrm{get}_C(n) = o \land n < \mathrm{next}_C)$$

for $n : \mathtt{int}$. The predicate states that the object o of class C is already created, which means that it is taken from the repository.

[1] This is only a design decision. There are logics, where the existence of all objects a priori is not required.

Given an object o of class C we would like to talk about the index of o in the repository of C and therefore define the abbreviation $\text{index}_C : C \to \texttt{int}$ as the function which returns the index $i : \texttt{int}$ for which holds

$$\text{get}_C(i) = o$$

given an object o of class C. If the object does not exist and thus there is no such index, the function returns some specific integer, which we do not define here any further.

Methods and Dynamic Dispatch

Classes (and objects) have methods to manipulate the state of the object. There are a variety of ways to represent methods in a logic. Because methods are actually not relevant for our work, we chose a rather simple way to deal with them. We assume that whenever the symbolic execution of the calculus (Section 8.3) encounters the invocation of a method, this invocation is replaced by the implementation of the method.

Because classes can inherit from each other, methods can be overwritten in subclasses of others. Therefore the right implementation has to be selected from the classes whenever it is inserted into a program at the point where it is invoked. This selection is called *dynamic dispatch*. It can be reduced to static method calls by a number of type-checks with instanceOf along the reverse order of the subtype relation.

Attributes

Attributes of objects store the states of objects. Like variables they are assigned a type. If an attribute is of type T, we write $a : T$. Attributes can be read or written in a program's execution. Objects in HEAP DL do not include attributes directly.

Attributes are represented as *non-rigid* functions. The values of non-rigid functions are not fixed from the beginning but can be manipulated in program statements and updates. Therefore they are useful to capture the notion of attributes. For each attribute a of a class C we introduce a non-rigid function symbol $a_C : C \to T$ if $a : T$. Reading access to the attribute of the object o of class C is then represented by the term $a_C(o)$. Writing access is written as $a_C(o) = m$ for example, where m is of the type of a.

As mentioned before, the syntax for the attributes of arrays does differ from this definition but we assume that the reader can abstract away from that.

Side-effects

The formal definition of HEAP programs, which we will give in Definition 8.6, demands that expressions in assignments or conditions are terms and as such side-effect-free. This is not an actual restriction of the set of HEAP programs, because programs with expressions which have side-effects can always be transformed to programs without those.

To simplify the calculus, we assume that all programs which contain expressions or assignments with side-effects are transformed like that. The transformation is done by introducing new program variables and replacing the side-effect-carrying assignments by a series of assignments to the new variables. Those assignments do not have effect on any other variable than the assigned one. [BHS07] describes this technique in detail in Section 3.6.2.

Some of the examples which we will examine in Section 8.4 might have side-effects in their expressions. This was not a problem, because we use the theorem prover KeY which performs the transformation of expressions which have side-effects by itself.

Exceptions

Heap programs do not include exceptions. This does not actually restrict the language because programs using exceptions can be transformed to programs without them. Statements which could throw exceptions can be surrounded by a series of checks on the critical variables for the respective events that would throw an exception[2].

8.2.2 Syntax of Heap DL

Classes of the object-oriented programming language Heap are represented as types in our logic. The inheritance relation between classes is mapped into the logic by introduction of a subtype relation on the types. The formal definition of the Heap DL type system[3] follows.

Definition 8.1 (Heap DL Types)**.** A Heap DL type system $T_{\textsc{Heap}} = (\mathcal{T}_{\textsc{Heap}}, \leqslant_{\textsc{Heap}})$ is a finite set of types $\mathcal{T}_{\textsc{Heap}}$ and a relation $\leqslant_{\textsc{Heap}}$ with the following properties.

- $\{\top, \bot\} \subseteq \mathcal{T}_{\textsc{Heap}}$, where $\top$ is called the *universal type* and $\bot$ is called the *empty type.*
- $\leqslant_{\textsc{Heap}}$ is a reflexive partial order on $\mathcal{T}_{\textsc{Heap}}$, i. e. for all types $A, B, C \in \mathcal{T}_{\textsc{Heap}}$,
  - $A \leqslant_{\textsc{Heap}} A$
  - if $A \leqslant_{\textsc{Heap}} B$ and $B \leqslant_{\textsc{Heap}} A$ then $A = B$
  - if $A \leqslant_{\textsc{Heap}} B$ and $B \leqslant_{\textsc{Heap}} C$ then $A \leqslant_{\textsc{Heap}} C$.
- A is called a *subtype* of B if $A \leqslant_{\textsc{Heap}} B$.
- $\bot \leqslant_{\textsc{Heap}} A \leqslant_{\textsc{Heap}} \top$ for all $A \in \mathcal{T}_{\textsc{Heap}}$.
- T is closed under greatest lower bounds with respect to $\leqslant_{\textsc{Heap}}$, i. e., for any $A, B \in \mathcal{T}_{\textsc{Heap}}$, there is an $I \in T$ such that $I \leqslant_{\textsc{Heap}} A$ and $I \leqslant_{\textsc{Heap}} B$ and for any $C \in \mathcal{T}_{\textsc{Heap}}$ such that $C \leqslant_{\textsc{Heap}} A$ and $C \leqslant_{\textsc{Heap}} B$, it holds that $C \leqslant_{\textsc{Heap}} I$. We

[2] The KeY prover can handle exceptions directly, but we do not include them in the logic, because we want to keep the calculus as small as possible.

[3] This definition is inspired by Definitions 2.1, 3.1 and 3.2 of [BHS07].

write $A \cap B$ for the greatest lower bound of A and B and call it the *intersection type* of A and B. The existence of $A \cap B$ also guarantees the existence of the least upper bound $A \cup B$ of A and B, called the *union type* of A and B.

- Then the *direct subtype relation* $\leqslant_0 \subseteq \mathcal{T}_{\text{HEAP}} \times \mathcal{T}_{\text{HEAP}}$ between two types $A, B \in \mathcal{T}_{\text{HEAP}}$ is defined as:

$$A \leqslant_0 B \text{ iff } A \leqslant_{\text{HEAP}} B \text{ and } A \neq B$$

and $C = A$ or $C = B$ for any $C \in \mathcal{T}_{\text{HEAP}}$ with $A \leqslant_{\text{HEAP}} C$ and $C \leqslant_{\text{HEAP}} B$.

- $\{\texttt{int}, \texttt{boolean}\} \subseteq \mathcal{T}_{\text{HEAP}}$ with $\bot \leqslant_0 \texttt{int} \leqslant_0 \top$, $\bot \leqslant_0 \texttt{boolean} \leqslant_0 \top$, $\texttt{int} \nleqslant \texttt{boolean}$ and $\texttt{boolean} \nleqslant \texttt{int}$.
- There is a type $\texttt{Null} \in \mathcal{T}_{\text{HEAP}}$ with $\bot \leqslant_0 \texttt{Null}$.
- There is a type $\texttt{Object} \in \mathcal{T}_{\text{HEAP}}$ with $\texttt{Object} \leqslant_0 \top$.
- $A \cap B = \bot$ for all $A \in \{\texttt{boolean}, \texttt{int}\}$ and $B \leqslant_{\text{HEAP}} \texttt{Object}$.
- If $A \leqslant_{\text{HEAP}} \texttt{Object}$, then $\texttt{Null} \leqslant_{\text{HEAP}} A$ for all $A \neq \bot \in \mathcal{T}_{\text{HEAP}}$.

Note that we leave out the subscript WHILE if it is clear about which version of the respective entity we are talking about.

A HEAP DL signature is similar to that of the WHILE language, except that there are two types of function symbols, rigid ones and non-rigid ones. The formal definition follows[4].

Definition 8.2 (HEAP DL Signature)**.** A HEAP DL *signature* for $\mathcal{T}_{\text{HEAP}}$ is a tuple $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}_r, \mathcal{F}_{nr}, \mathcal{P}, \alpha)$. $\mathcal{V}_l, \mathcal{V}_p, \mathcal{P}, \alpha$ are defined as in Definition 3.2 for WHILE DL, except that here they are based on a HEAP DL type system $\mathcal{T}_{\text{HEAP}}$ instead of $\mathcal{T}_{\text{WHILE}}$. $\mathcal{F}_r$ is a set of rigid function symbols and $\mathcal{F}_{nr}$ is one of non-rigid function symbols. α has the following property concerning $\mathcal{F}_r$ and $\mathcal{F}_{nr}$: $\alpha(f) \in \mathcal{T}^n_{\text{HEAP}} \times \mathcal{T}_{\text{HEAP}}$ for all $f \in \mathcal{F}_r \cup \mathcal{F}_{nr}$ of arity n. The set of function symbols $\mathcal{F}_r$ contains the elements of $\mathcal{F}$ defined in Definition 3.2 and additionally:

- The object repository access function

$$\text{get}_A : \texttt{int} \to C$$

for any $A \in \mathcal{T}_{\text{HEAP}} \backslash \{\bot, \texttt{Null}, \texttt{int}, \texttt{boolean}, \top\}$.

- The literal (nullary function symbol) $\texttt{null}$ of type $\texttt{Null}$.

The set of non-rigid function symbols $\mathcal{F}_{nr}$ contains the following elements.

- The object enumeration function

$$\text{next}_C : \to \texttt{int}$$

for each type $C \in \mathcal{T} \backslash \{\bot, \texttt{Null}, \texttt{int}, \texttt{boolean}, \top\}$.

- An attribute function $a_C : C \to T$ for each attribute $a : T$ of class C.

[4] This definition is inspired by Definition 3.4 of [BHS07] and the handling of attributes in [BP06].

We use the notation $f : A_1, \ldots, A_n \to A$ for $\alpha(f) = ((A_1, \ldots, A_n), A)$ and $f \in \mathcal{F}_r \cup \mathcal{F}_{nr}$.

Terms and updates in Heap DL are defined as they are in While DL with the adaptation to non-rigid function symbols. Those symbols can be part of a term and in updates they can be assigned new values. The definition of terms uses the definition of updates and vice versa. This is not a mistake, but a deliberate decision made to make the concept as intuitively understandable as possible[5].

Definition 8.3 (Heap DL Terms)**.** Given a Heap signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}_r, \mathcal{F}_{nr}, \mathcal{P}, \alpha)$ for a type system T_{HEAP}, we inductively define the system of sets $\{\mathfrak{T}_{\text{HEAP},A}\}_{A \in \mathcal{T}_{\text{HEAP}}}$ of terms of type A the same way as we did for While DL in Definition 3.6 with the exception of the following adaptations:

- $\mathcal{F}$ is replaced by $\mathcal{F}_r \cup \mathcal{F}_{nr}$, and
- all other entities (for example updates) refer to their respective Heap DL version.

$\sigma : \mathfrak{T}_{\text{HEAP}} \to \mathcal{T}_{\text{HEAP}}$ is the function which returns the type A of each term $t \in \mathfrak{T}_{\text{HEAP},A}$.

In addition to the updates of non-rigid function symbols, we introduce another type of updates, called quantified updates. A quantified update describes the manipulation of a an unbounded number of updates. This is in particular useful when talking about datastructures which are of a finite but unknown length as for example linked lists.

Definition 8.4 (Heap DL Updates)**.** Let $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}_r, \mathcal{F}_{nr}, \mathcal{P}, \alpha)$ be given as a Heap DL signature for the type set $\mathcal{T}_{\text{HEAP}}$, the set $\mathfrak{U}_{\text{HEAP}}$ of *syntactic updates* is inductively defined as the least set such that:

- $(v := t) \in \mathfrak{U}_{\text{HEAP}}$ and $(u_1 \mid\mid u_2) \in \mathfrak{U}_{\text{HEAP}}$ as defined in Definition 3.5
- $(f(t_1, \ldots, t_n) := t) \in \mathfrak{U}_{\text{HEAP}}$ for all terms $f(t_1, \ldots, t_n) \in \mathfrak{T}_A$ with $f \in \mathcal{F}_{nr}$ and $t \in \mathfrak{T}_{\text{HEAP},A'}$ such that $A' \leqslant_{\text{HEAP}} A$ (Function update)
- $(\forall x \; \varphi \; u) \in \mathfrak{U}_{\text{HEAP}}$ for all $u \in \mathfrak{U}_{\text{HEAP}}, x \in \mathcal{V}$ and $\varphi \in \mathfrak{F}$ (Quantified update)

Ground terms are defined in Heap DL exactly like in While DL (Definition 3.8) which is why we do not repeat the definition here. The definition of rigid terms has to take the absence of non-rigid function symbols into account[6].

Definition 8.5 (Rigid Terms)**.** A Heap DL term t is *rigid*, if it complies to Definition 3.9. In particular, t does not contain non-rigid function symbols $f \in \mathcal{F}_{nr}$.

Heap programs have the same constructs as While programs (Definition 3.10), except that variables point to objects and thus have attributes and the respective Heap DL types. Attributes can be assigned new values and thus these assignments have to be included in the definition of programs.

[5] The definitions of Heap DL terms and updates are inspired by [BHS07], Definition 3.7 and 3.8.

[6] This definition is taken from [BHS07], Definition 3.31.

Definition 8.6 (Heap Programs)**.** The set of programs $\mathfrak{P}_{\text{Heap}}$ is inductively defined the same way as $\mathfrak{P}_{\text{While}}$ (Definition 3.10), except that all entities are based on their respective Heap DL version and with the following additional element.

- $f(t_1, \ldots, t_n) = t; \in \mathfrak{P}_{\text{Heap}}$, where $f \in \mathcal{F}_{nr}$ and $t \in \mathfrak{T}$ a ground term of the same type as the output type of f and $t_1, \ldots, t_n$ ground terms.

In the example programs in this thesis, the program statement `o = new C()` for some variable o and some class C can occur. This is the creation of an object. We do not introduce such a statement in the logic. We rather assume that the statement is an abbreviation for the statement[7]

$$o := \text{get}_C(\text{next}_C) \mid\mid \text{next}_C := \text{next}_C + 1.$$

Finally, we have to adapt the definitions of formulae of While DL to make them suitable for Heap DL.

Definition 8.7 (Heap DL Formulae)**.** Let a signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}_r, \mathcal{F}_{nr}, \mathcal{P}, \alpha)$ for the type system T_{Heap} and a Heap program p be given. Then the set $\mathfrak{F}_{\text{Heap}}$ of Heap DL formulae is defined the same way as While DL formulae except that the used entities are of the respective Heap DL versions, for example $\mathfrak{P}_{\text{Heap}}$ instead of $\mathfrak{P}_{\text{While}}$.

Definition 8.8 (Rigid Formulae)**.** A *rigid* Heap DL formula φ is defined the same way as rigid While DL formulae except that all used entities are of their respective Heap versions.

Definition 8.9 (Free Variables in Heap DL)**.** We define the set $\text{fv}(u)$ of free variables of an update u for variable updates and parallel updates the same way as for While DL except that the occurring entities are of their respective Heap versions. Additionally, we define the free variables of a function update as

$$\text{fv}(f(t_1, \ldots, t_n) := t) = \text{fv}(t) \cup \bigcup_{i=1\ldots n} \text{fv}(t_i).$$

We define $\text{fv}(t)$, the set of free variables of a term t the same way as for While terms except that

$$\text{fv}(f(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} \text{fv}(t_i)$$

is defined for both types of function symbols, which means for $f \in \mathcal{F}_r \cup \mathcal{F}_{nr}$.
We define $\text{fv}(\varphi)$, the set of free variables of a formula φ, exactly the same way as for While formulae, except that all used entities are in their respective Heap entities.

[7] We assume that this transformation is done for the program, because it keeps the number of program statements and thus the definition of the logic and the calculus as simple as possible.

8.2.3 Semantics of Heap DL

We define the semantics for Heap DL similar to the one for While DL. Note that also in Heap DL models, the interpretation function $\mathcal{I}_{\text{Heap}}$ assigns functions only to rigid function symbols. The semantics of non-rigid function symbols is given as function assignment later in this section.

Definition 8.10 (Heap DL Model)**.** Given a Heap DL type system T_{Heap} and a Heap DL signature, a Heap *model* is the triple $\mathcal{M}_{\text{Heap}}(\mathcal{D}_{\text{Heap}}, \delta_{\text{Heap}}, \mathcal{I}_{\text{Heap}})$ with the following properties.

- δ_{Heap} is a type function

$$\delta_{\text{Heap}} : \mathcal{D}_{\text{Heap}} \to \mathcal{T}_{\text{Heap}} \backslash \{\bot\},$$

- The *domain* $\mathcal{D}_{\text{Heap}}$, is defined as

$$\mathcal{D}_{\text{Heap}} = \bigcup_{A \in \mathcal{T}_{\text{Heap}}} \mathcal{D}_A$$

  with

$$\mathcal{D}_A = \{d \in \mathcal{D}_{\text{Heap}} \mid \delta_{\text{Heap}}(d) \leqslant A\}$$

  and the fixed domains

$$\begin{aligned} \mathcal{D}_{\texttt{int}} &= \{\dots, -2, -1, 0, 1, 2, \dots\}, \\ \mathcal{D}_{\texttt{boolean}} &= \{\mathit{true}, \mathit{false}\}, \text{and} \\ \mathcal{D}_{\texttt{Null}} &= \{\mathit{null}\}. \end{aligned}$$

  Each domain $\mathcal{D}_A$ for $A \in \mathcal{T}_{\text{Heap}} \backslash \{\texttt{Null}, \texttt{int}, \texttt{boolean}, \bot, \top\}$ contains countably infinite many distinct domain elements.

- The *interpretation* $\mathcal{I}_{\text{Heap}}$ maps each rigid function symbol $f : A_1, \dots, A_n \to A \in \mathcal{F}_r$ to a function

$$\mathcal{I}_{\text{Heap}}(f) : \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n} \to \mathcal{D}_A$$

  and each predicate symbol $p : A_1, \dots, A_n \in \mathcal{P}$ to a subset

$$\mathcal{I}_{\text{Heap}}(p) \subseteq \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n},$$

  with $A_1, \dots, A_n, A \in \mathcal{T}$.

$\mathcal{I}_{\text{Heap}}$ maps integer and boolean literals and operations as $\mathcal{I}_{\text{While}}$ does. Furthermore we demand that $\mathcal{I}_{\text{Heap}}(\texttt{null}) = \mathit{null}$. $\mathcal{I}$ maps the repository access function symbol get_C to the actual function, which maps an object o to its index, the non-negative integer i for which the equation $\mathcal{I}(\text{get}_C)(i) = o$ holds. For negative integers get_C is also defined, but its values are unknown.

As we mentioned in Section 8.2.1, there is an object repository[8] for each class C.

[8] The definition is taken from [BHS07], Definition 3.52.

Definition 8.11 (Object Repository)**.** Given a type $C \in \mathcal{T}_{\textsc{Heap}}$, the *object repository* Rep_C is the set of all domain elements e of type C:

$$\mathrm{Rep}_C := \{e \in \mathcal{D}_{\textsc{Heap}} \mid \delta_{\textsc{Heap}}(e) = C\}$$

The difference between the domain $\mathcal{D}_A$ of a type A and the repository Rep_A is that the latter contains the elements of type A only and *not* the ones of A's subtypes.

We defined logical variable assignments for While DL in Definition 3.19. For Heap DL we define it exactly the same way, except that the used entities are of their respective Heap DL versions, for example a Heap DL model instead of a While DL model.

In Heap DL not only program variables can be assigned values, but also non-rigid functions can be manipulated in programs. We embrace those two in the set of locations.

Definition 8.12 (Location)**.** Given a Heap DL signature and a model, the set $\mathcal{L}$ of locations is defined as

$$\begin{aligned} \mathcal{L} = \quad \mathcal{V}_p \cup \{(f, d_1, \dots, d_n) \mid \quad & f \in \mathcal{F}_{nr} \text{ of arity } n, \\ & f : A_1, \dots, A_n \to A, \\ & d_i \in \mathcal{D}_{A_i} \text{ for } i = 1, \dots, n\} \end{aligned}$$

As a convention, we write $\delta(f) = A$ for $f : A_1, \dots, A_n \to A$ so that δ is defined for all $l \in \mathcal{L}$.

Therefore we extend the notion of program variable assignments of While DL (Definition 3.21) by defining the function γ not only for program variables but for the set $\mathcal{L}$.

Definition 8.13 (Location Assignment)**.** Given a model $\mathcal{M}_{\textsc{Heap}} = (\mathcal{D}_{\textsc{Heap}}, \mathcal{I}_{\textsc{Heap}})$, a *location assignment* is a function $\gamma_{\textsc{Heap}}$ such that

$$\gamma_{\textsc{Heap}}(l) \in \mathcal{D}_A \text{ for } l \in \mathcal{L}_{\mathcal{M}_{\textsc{Heap}}} \text{ with } \delta(l) = A$$

A Heap program's state space is then defined similar to the one of While programs, except that the extended notion of γ is used here.

Definition 8.14 (Program State and State Space)**.** Given a model $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$, the set $\mathcal{S}^{\mathcal{M}}$ of program states is defined as

$$\mathcal{S}^{\mathcal{M}} = \{\gamma \mid \gamma : \mathcal{L}_{\mathcal{M}} \to \mathcal{D}\}$$

The extended state space $\mathcal{S}^{\mathcal{M}}_{\infty}$ is defined analogous to $\mathcal{S}^{\mathcal{M}}_{\infty}$ for While DL.

Definition 8.15 (Semantics of Terms)**.** Let $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ be a model, β a logical variable assignment and γ a location assignment. We inductively define the valuation function $\mathrm{val}_{\mathcal{M},\beta,\gamma}$ for terms the same way as for terms in While DL (Definition 3.24), with the addition of the definition of non-rigid functions:

- $\text{val}_{\mathcal{M},\beta,\gamma}(f(t_1,\ldots,t_n)) = \gamma(f, \text{val}_{\mathcal{M},\beta,\gamma}(t_1),\ldots,\text{val}_{\mathcal{M},\beta,\gamma}(t_n))$ for every $f \in \mathcal{F}_{nr}$ and $t_i \in \mathfrak{T}$.

The semantics of updates and terms corresponds to their versions in While DL except that the non-rigid functions and quantified updates are considered. Like in the definitions of their syntax, the two definitions depend on each other.

The exact definition of the semantics of quantified updates is rather complex. We abstain from quoting it here because of space limitations. Intuitively, a quantified update $(\forall x\ \varphi\ u)$ applies the update u to all locations x which fulfill the formula φ. For a detailed explanation of the semantics of quantified updates see [Rüm06], Section 4.

Definition 8.16 (Semantics of Updates)**.** Updates $u \in \mathfrak{U}$ are interpreted as partial functions from the state space to partial location assignments

$$[\![u]\!]^{\mathcal{M},\beta} : \mathcal{S}^{\mathcal{M}} \to (\mathcal{L}_{\mathcal{M}} \twoheadrightarrow \mathcal{D}),$$

where $\twoheadrightarrow \mathcal{D}$ stands for a partial function to the domain $\mathcal{D}$. The semantics for single and parallel updates is defined the same way as for While DL updates. The semantics of function updates then is

$$[\![f(t_1,\ldots,t_n) := t]\!]^{\mathcal{M},\beta}(\gamma)(x) \ := \ \{(f, \text{val}_{M,\beta,\gamma}(t_1),\ldots,\text{val}_{M,\beta,\gamma}(t_n)) \mapsto \text{val}_{\mathcal{M},\beta,\gamma}(t)\}\,.$$

Definition 8.17 (Semantics of Programs)**.** Given a $\mathcal{M} = (\mathcal{D}, \mathcal{I})$, a program is a function:

$$[\![p]\!]^{\mathcal{M}} : \mathcal{S}^{\mathcal{M}}_{\infty} \to \mathcal{S}^{\mathcal{M}}_{\infty}.$$

The semantics of Heap DL programs are defined the same way as of While programs (Definition 3.25), except that the used entities are of the respective Heap DL versions. Additionally, we define the semantics for function assignments as follows.

$$[\![f(t_1,\ldots,t_n) = t]\!]^{\mathcal{M}}(\gamma)(x) = \begin{cases} \text{val}_{\mathcal{M},\gamma}(t) & \text{if } x = (f, \text{val}_{\mathcal{M},\gamma}(t_1),\ldots,\text{val}_{\mathcal{M},\gamma}(t_n)) \\ \gamma(x) & \text{otherwise.} \end{cases}$$

Finally, we can define the semantics of Heap DL formulae.

Definition 8.18 (Semantics of Formulae)**.** The semantics of Heap DL formulae are defined the same way as While DL formulae (Definition 3.29) with the adaptation that all used entities are of their respective Heap versions.

8.3 Heap DL Calculus

We extend the While calculus, which we presented in Section 3.3, to make it suitable for Heap DL formulae. The essential changes are the handling of the more complex type system, object creation and attribute manipulation.

The definitions of sequent formulae and proof trees for Heap DL calculus are the same as for While DL, except that the occurring formulae are Heap DL formulae

$$\frac{}{\Gamma \Rightarrow \mathrm{get}_C(i) = \mathrm{get}_C(j) \rightarrow i = j, \Delta}\ \textsf{closeSameIdx}$$

$$\frac{}{\Gamma \Rightarrow \neg(\mathrm{get}_C(i) = \mathrm{get}_D(j)), \Delta}\ \textsf{closeClassDiff}$$

$$\frac{}{\Gamma \Rightarrow \mathrm{get}_C(i) \not\doteq \texttt{null}, \Delta}\ \textsf{closeNull}$$

$$\frac{}{\Gamma \Rightarrow \forall o : C(\bigvee_{D \leqslant C}(\exists i\ \texttt{int}\ \mathrm{get}_D(i) = o) \vee o = \texttt{null}), \Delta}\ \textsf{closeExistsOrNull}$$

Figure 8.2: Class Rules of the Heap Calculus

instead of While DL ones. We include the classical first-order rules which we showed in Figure 3.3 in our calculus in exactly the same form as in While DL.

The closing rules shown in Figure 3.4 are included in the Heap calculus as well as all equality rules of Figure 3.5.

The calculus rules concerning symbolic execution of programs can widely be used in the Heap DL calculus as well. This applies to the modality rules in Figure 3.6 and to the assignment and conditional rules in Figure 3.7. The latter two can be included into the Heap calculus, because we assume that expressions which have side-effects are transformed into side-effect-free ones, otherwise the calculus would be more complex.

We access the object repository via the function get_C. This way there are a few more rules which enable us to close a proof. See the rules in Figure 8.2.

Like for the While DL calculus we do not explicitly mention rules for handling arithmetic or the application of updates here. We refer to the respective references in Section 3.3. We include the rules to introduce fresh metavariables of Figure 3.12 in the calculus, because also for heap programs our algorithm applies the technique of closing proofs by constraints (Section 3.4).

Loop rules

Concerning the calculus rules which handle loops in programs, we include the rules loopUnwind (Figure 3.8) in our calculus. We also add the improved invariant rules invRuleMod and invRuleTermMod to the calculus (Figure 3.10), but we have to adjust the definition of modifier sets to make them suitable for our logic, because in Heap programs not only program variables but also attributes can be modified.

Definition 8.19 (Syntax of Modifier Sets)**.** Let $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$ be a signature for a type system T_{HEAP}. A modifier set Mod is a subset of $\mathcal{V}_p \cup \mathcal{F}_{nr}$.

This definition is actually quite coarse. A modifier set that contains a non-rigid function symbol a_C basically says that the attribute a of any object of class C might be changed in the program's execution. It is possible to define modifier sets more specific (see for example Definition 3.61 in [BHS07]), but for our purpose this definition is sufficient.

In the application of the invariant rule invRuleMod for each element in a modifier set an anonymizing update is created. Because non-rigid function symbols in the modifier set make a statement about a manipulation of all possible objects of the class, we have to form a quantified update, which anonymizes the attribute for all objects of this class. This is the reason why we introduced the quantified update in Definition 8.4. Thus an attribute a of class C in the modifier set leads to the following quantified update.

$$\forall o\ true\ a_C(o) := a_C(o)^*(X_1, \ldots, X_n)$$

where $a_C(o)^*$ is a fresh function symbol for the respective attribute of each object and $X_1, \ldots, X_n$ are the collected metavariables as described in Definition 3.40.

Definition 8.20 (Semantics of Modifier Sets)**.** Given a signature $(\mathcal{V}_l, \mathcal{V}_p, \mathcal{F}, \mathcal{P}, \alpha)$, let $\mathcal{M} = (\mathcal{D}, \delta, \mathcal{I})$ be a model, β a logical variable assignment and p a program. A pair of states $(\gamma_1, \gamma_2) = \mathcal{S}^{\mathcal{M}} \times \mathcal{S}^{\mathcal{M}}$ *satisfies* a modifier set Mod

$$(s_1, s_2) \models \text{Mod}$$

iff, for all $v \in \mathcal{V}_p$ the following holds

$$\gamma_1(v) \neq \gamma_2(v) \text{ implies that } v \in \text{Mod},$$

and for all $t = f(d_1, \ldots, d_n)$ with $f \in \mathcal{F}_{nr}$ and $d_1, \ldots, d_n \in \mathcal{D}_{\text{HEAP}}$ holds

$$\gamma_1(f, d_1, \ldots, d_n) \neq \gamma_2(f, d_1, \ldots, d_n) \text{ implies that } f \in \text{Mod}\,.$$

The modifier set Mod is *correct* for the program p, if

$$(\gamma_1, \gamma_2) \models \text{Mod}$$

for all state pairs (γ_1, γ_2), where γ_1 is the start state of p and γ_2 the result state of it.

A minimal modifier set of a program contains only the variables and attributes which are actually changed in the program and nothing more.

Definition 8.21 (Minimal modifier set)**.** Let p be a program. A *minimal modifier set* Mod_p of a program p does not contain any $v \in \mathcal{V}_p$ or $f \in \mathcal{F}_{nr}$ for which

$$\gamma_1(v) = \gamma_2(v)$$

or

$$\gamma_1(f, d_1, \ldots, d_n) = \gamma_2(f, d_1, \ldots, d_n)$$

holds for some $d_i \in \mathcal{D}_{\text{HEAP}}$ and for all pairs of states $(\gamma_1, \gamma_2) \in \mathcal{S} \times \mathcal{S}$, where γ_1 is a start state of the program and γ_2 is the result state of the program.

With these extended versions of the modifier set, the definitions of the anonymizing updates (Definition 3.40) and anonymizing updates with respect to a modifier set (Definition 3.40) are the same as for WHILE DL except that the new definitions of updates are used.

The invariant rules invRuleMod and invRuleTermMod, which use anonymizing updates, are then the same for HEAP DL except that we assume that the used updates comply to the definitions of this section.

8.4 Non-termination Analysis of Examples

In this section, we will examine four examples of HEAP programs with respect to their termination behavior. We applied our algorithm on them to see how well our method can be transfered to this programming language. We did not use our software for the experiments, because we designed it for WHILE programs in the first place and thus it is not fully capable of dealing with complex data structures in open goals.

However, we manually did the steps which the invariant generator does and invoked the KEY prover in the point of the algorithm where the invariant generator would invoke it. The KEY version which we used for these examples was version 76 of the COUNTEREXAMPLES branch.

In the evaluation we did not follow any of the particular heuristics of the ones described in Chapter 5. We rather assumed that there are “perfect” heuristics which do have the intuition which we have in order to pick the useful open goals. We assume that it is always possible to adjust heuristics in a way that they work like in these examples. Thus we examined in these examples *if* it is possible to construct a useful invariant rather than for which heuristics the invariant would be found.

We will now present four examples ARRAYSUM, CHAOSBUFFER, TRAVERSE and TAKESHI. ARRAYSUM is a program which uses array data structures but the non-termination invariant is actually not dependent on the contents of the array. CHAOSBUFFER uses also an array and here the content of the array is essential for the non-termination of the program. TRAVERSE is an example of a program where a linked list as data structure is used. TAKESHI is a more advanced example using linked lists, because here the form of the list is changed during loop execution. We describe what preparations have been necessary to examine the examples with the KEY prover and evaluate how well our algorithm worked on them.

8.4.1 Note on Abrupt Termination

In Section 4.1, we explained that the formula $[p]$*false* states the non-termination of a program p. This is actually only correct for programs which cannot terminate abruptly. Abrupt termination means that an exception occurs and the program instantly dies. The formula $[p]$*false* is also fulfilled if the program p terminates abruptly. This means that the logic does not make a difference between non-termination and abrupt termination. This is a reasonable definition, because both behaviors are unappreciated.

By definition, exceptions cannot occur in WHILE programs. In HEAP programs we officially demand that statements which can throw an exception are transformed into a program fragment, where the critical statement is surrounded by checks for all possibly critical situations. The programs which we will examine in the following sections do actually contain statements which can cause exceptions. Because we like to maintain the readability of the programs we did not transform them into exception-free programs.

If we apply our algorithm on our examples without regarding the possibility of an exception, it sometimes does not find input values which lead to non-termination but values which make the program crash with an exception. The latter is also good if we are looking for bugs in general, but not if we are especially interested in detecting non-termination.

To avoid these distracting discoveries of abrupt-termination-bugs, we apply a trick. Instead of stating the non-termination in a formula like $[p]\mathit{false}$, we surround our example programs (which are written in JAVA syntax) with a `try-catch` block.

$$[\texttt{try}\ \{\ p\ \}\ \texttt{catch}\ (\texttt{Exception e})\{\ \}]\mathit{false}$$

This way, all exceptions are caught and thus the program does terminate in a normal way. If our algorithm can close the proof of this formula it is because of non-termination and not because of abrupt termination. We applied this additional preparation in some of the examples if it was necessary.

8.4.2 Example ARRAYSUM

The code of the example program ARRAYSUM is shown in Figure 8.3. The purpose of the program is to sum up the elements of the array of integers which is given as input. The summing is done in a `while` loop over the index of the array elements. The programmer has made a typical programming error; he forgot to increase the running variable `i` in the loop body.

Preparations

For the application of our algorithm on this example we did not have to change the code of the program itself. We wrote a KEY-file to give the prover some necessary information about the data structures which were used.

The first preparation is the introduction of the input parameters. In this case, it is the array `int[] a`, which we specify as program variable. Like for WHILE programs, we have to existentially quantify over the input variables. For an array this is more complicated than for single variables, because we do not only have to quantify over all possible arrays, but also over all possible lengths of arrays and all possible contents of arrays. For all these we have to introduce metavariables.

Because we have to quantify over all contents of all possible arrays, we have to introduce a metavariable of a matrix-like type, containing metavariables for each

```
arraysum(int[] a) {
  int sum = 0;
  int i = 0;
  while (i < a.length) {
    sum += a[i];
  }
  return sum;
}
```

Figure 8.3: ArraySum

This program sums up the contents of an array of integers. It does not terminate, because the programmer forgot to increase the running variable `i` in the loop body.

entry of each array. Unfortunately, the KeY language does not support this feature so far. Therefore we assume that we are talking about one particular array, whose name we already know and whose properties we are going to find out in the generation of the invariant; in our case it is the array `a`. Because now we only talk about one array, it is sufficient to introduce a metavariable of the type of a list for all possible contents of this array.

The list in our case i called `ACONTS` and to specify it as a list, we also have to define what a list is and what properties it has. The KeY language provides syntactical elements for that. We will not present all details here, because it is basically the usual recursive definition of lists. Additionally, we introduce a metavariable `ALEN` for the length of `a`.

The actual proof obligation is written as follows. We introduced the precondition `a` $\not\doteq$ `null` here. As mentioned before, we do not quantify over all possible arrays. Therefore we have ensure that the specific array we are talking about is not `null`. If this precondition was not fulfilled it would make no sense to look for a non-terminating loop, because the program would crash with an exception anyway. The complete proof obligation looks like the following.

$$\begin{gathered} \texttt{a} \not\doteq \texttt{null} \rightarrow \\ \{\texttt{length}_{\texttt{Array}}(a) := \texttt{ALEN},\ \forall \texttt{k}; \texttt{a[k]} := \texttt{alInt(ACONTS, k)}\} \\ [\texttt{try}\ \{\ \texttt{arraySum(a);}\ \}\ \texttt{catch}\ \{\ \}\ ]\mathit{false} \end{gathered}$$

The update in this formula introduces the respective metavariables, where `alInt` is the access to the metavariable list which assigns a metavariable for each element of the array.

For the application of the invariant rule invRuleMod, we need a modifier set. For this program the set {`sum`} is a minimal modifier set. If the programmer of this program had determined the modifier set himself, he would have noticed that the running variable `i` is not changed in the program, which is actually the bug of this program.

Application of the algorithm

We apply the algorithm as it is described in Chapter 5 on this example. We will not show the proof trees here, but examine the results of the proofs of each iteration.

Iteration no 1. With invariant candidate $inv_1 \equiv true$ in the first iteration the prover can close the proof with the following constraint.

$$0 < \texttt{ALEN},$$

It says that the array must at least contain one element.

Evaluation

The application of the algorithm on this simple example gives reason to be optimistic that the developed method is applicable to more complex programs than just While programs. The program contains an array data-structure, but the invariant is actually not dependent on the contents of the array. We suppose that most of the programs which have this property could be handled by our algorithm, because the form of the invariant has the same level of complexity as invariants of While programs.

8.4.3 Example ChaosBuffer

Figure 8.1 shows the code of the program ChaosBuffer[9]. The chaos buffer is a class which has an array and an attribute `cursor` which is an integer pointing to a place in the array. When a new value is stored in the array, the buffer goes through the array starting with the place where the cursor points to until it finds an element in the array which is `0`. At this place, the new value is stored. If the buffer reaches the end of the array in the search for a free place, the cursor is set to `0` and the search goes on from there.

The programmer of this program forgot to think about the situation when no entry of the array is equal to zero. In this case the cursor runs through the array over and over again, desperately looking for a free spot in the array. We applied our algorithm on the function `insertInput` which performs the insertion of a new element.

Preparations The preparations of this example are similar to the ones for the example ArraySum. The data structure of a list is described in the logic and for all input parameters natural metavariables are assigned. The proof obligation

[9] We thank Juri Ganitkevitch for the contribution of this example.

contains the precondition $\mathtt{a} \doteq \mathtt{null}$, which has the same purpose as in the example ArraySum. Thus the proof obligation for this example is the following.

$$\begin{array}{l}
\mathtt{a} \doteq \mathtt{null} \rightarrow \\
\\
\{\ \mathtt{length}_{\mathtt{Array}}(\mathtt{a}) := \mathtt{ALEN}, \\
\ \ \forall \mathtt{k}\ \mathtt{a[k]} := \mathtt{alInt\ (ACONTS,\ k)}, \\
\ \ \mathtt{n} := \mathtt{NEW}, \quad \mathtt{c} := \mathtt{CURS}\ \} \\
[\ \mathtt{try}\ \{\ \mathtt{insertInput(n,a,c);}\ \}\ \mathtt{catch}\ \{\ \}]\, \mathit{false}
\end{array}$$

A minimal modifier set for this program is $\{\mathtt{cursor}\}$.

Iteration No. 1 With the invariant candidate $\mathit{inv}_1 \equiv \mathit{true}$ the prover can construct a proof which has several open goals. One of them is the following.

$$\mathtt{cursor} \geqslant \mathtt{ALEN} \Rightarrow \mathtt{a} = \mathtt{null}$$

The formula $\mathtt{a} = \mathtt{null}$ originates from the precondition and thus is not useful for the invariant refinement. We use the negation of the formula $\mathtt{cursor} \geqslant \mathtt{ALEN}$ for the new invariant candidate and obtain:

$$\mathit{inv}_2 \equiv \mathit{true} \wedge \mathtt{cursor} < \mathtt{ALEN}$$

Iteration No. 2 With the invariant candidate inv_2, we obtain again a number of open goals of which one is the following.

$$\mathtt{cursor} \leqslant -1, \mathtt{cursor} \leqslant -1 + \mathtt{ALEN} \Rightarrow \mathtt{a} = \mathtt{null}$$

The formula $\mathtt{cursor} \leqslant -1 + \mathtt{ALEN}$ is not useful for invariant refinement, because its negation describes the situation where the cursor is out of the range of the array and thus an exception would occur. Therefore, the only interesting formula is $\mathtt{cursor} \leqslant -1$ whose negation we add to the invariant candidate and obtain candidate inv_3.

$$\mathit{inv}_3 \equiv \mathit{true} \wedge \mathtt{cursor} < \mathtt{ALEN} \wedge \mathtt{cursor} > -1$$

Iteration No. 3 With invariant candidate inv_3 the proof can be closed with the following constraints[10].

$$\mathtt{cursor} < \mathtt{a.length} \wedge -1 < \mathtt{cursor} \wedge \mathtt{a.length} < 2 \wedge \mathtt{a[0]\ \ !=\ 0}$$

This constraint says that the array has only one element and this one is not equal 0. This describes the smallest possible counter example for the termination of the program.

[10] We simplified the constraints and retranslated them into terms of the program's data structure to make them more human readable.

```
class Node {
  int value;
  Node succ;

  // constructor
  Node(int newValue) {
    this.value = newValue;
  }
}
```

```
traverse(Node head) {
  Node run = head;
  while (run != null) {
    run = run.succ;
  }
}
```

Figure 8.4: Traverse

This program is a simple traversal of a linked list. It does not terminate if the input list is cyclic.

Evaluation

The non-termination of the problem can be solved using our approach. The algorithm constructed the smallest possible counter example. This program shows that the search for the solution is highly dependent on the constraint solver, because in this example the constraints contribute an essential part of the information about the counter example. Finding such small counter examples is very useful for the debugging process because it shows the relevant aspects for this non-termination but without a complex specification of the example itself.

8.4.4 Example Traverse

The program Traverse is shown in Figure 8.4. The used data structure is a linked list and the action that is performed in the program is a basic list traversal as it implemented in many standard list operations.

The program looks completely correct at first sight; it is written as most programmers would write it. The problem of non-termination occurs when the list whose starting node is given as input contains a cycle. In this case none of the nodes has the value `null` in its attribute `succ`, which means that the loop condition is always true and thus the loop never terminates.

Preparation

As for the array examples, we have to prepare a KeY-file for the prover. We define the datastructure of a list of integers in the logic and introduce metavariables for all possible input values. The definition of the list is done as in the example ArraySum. The input parameter of the program is the head of the list.

In contrast to example ArraySum, we do not assume that we have only one list (respectively array) whose properties we have to define, but instead we quantify over all possible lists. The introduced metavariables are `idHead` for the position of the head in the list and `kNodes` for the length of the list. Because we quantify over all possible lists, we do not have preconditions in the proof obligation.

We specify that the position of the `head` node is within range, which means between 1 and the length of the list. This step is only for shortening the proof, because if it was out of range the proof would not be closed anyway, because the program would crash with an exception and thus terminate anyway (see the note on abrupt termination in Section 8.4.1).

$$\begin{gathered}\Rightarrow \texttt{idHead} \geqslant 0 \wedge \texttt{idHead} < \texttt{kNodes} \;\wedge \{\ldots\}\\ (\texttt{a} \neq \texttt{null} \rightarrow [\texttt{try \{ traverse(a); \} catch} \;(\texttt{Exception e})\; \{\;\}\;]\,\mathit{false})\end{gathered}$$

We left out a quite complex update in the obligation. It basically specifies the access to the list. We abstain from explaining all details here, because some of the definitions are KeY specific and their semantics are space consuming to explain. A minimal modifier set for this program is the set $\{\texttt{run}\}$.

Application of the algorithm

Iteration no. 1 We started the algorithm with the first invariant candidate

$$\mathit{inv}_1 \equiv \texttt{run} = \texttt{null} \vee \mathrm{created}_{\texttt{Node}}(\texttt{run}).$$

Apparently, we did not start the algorithm with the usual initial invariant *true*. The reason is a slight difference between the dealing with objects in Java and in Heap DL. The invariant above looks like a tautology in the first place, because in Java a variable which does not point to an object which is already created is always `null`. The problem is that technically in Heap DL the situation can be expressed that a variable points to an object which is not created yet, that means that this objects index is higher than the respective next_C counter is. This situation can never occur in Java programs and thus we include this information in the invariant candidate from the beginning. Thus, the formula makes sure that no unreachable states are considered in the proof.

The results of the iteration of the algorithm with invariant candidate inv_1 is an open proof with several open goals. Because of space limitation, we only pick the interesting formula, which occurs in the antecedent of one of the open goals.

$$\texttt{run} = \texttt{null}$$

We form the new invariant candidate inv_2 with the negation of this formula.

$$\mathit{inv}_2 \equiv (\texttt{run} = \texttt{null} \vee \mathrm{created}_{\texttt{Node}}(\texttt{run})) \wedge \texttt{run} \neq \texttt{null}$$

Iteration no. 2 In the second iteration of the algorithm, the prover can close the proof with the following constraints[11].

$$\texttt{idHead} = 0 \wedge \texttt{succ}_{\texttt{Node}}(\texttt{run}) = \texttt{run} \wedge \texttt{kNodes} = 1$$

[11] We simplified the constraint to make it more human readable.

The constraint says that the list is only one element long and that run points the first element and that its succ attribute points to run itself. The invariant can be simplified to

$$inv_2 \equiv \text{created}_{\texttt{Node}}(\texttt{run}) \land \texttt{run} \neq \texttt{null},$$

which means that it only says that the node run actually has to be created.

Evaluation

This example shows that the algorithm is also capable of working on programs with a linked list as data structure. The algorithm did find a counter example for the termination without human interaction in the proof construction. The preparations of the proof obligation and the creation of the slightly enhanced initial invariant candidate could be done automatically as well.

The algorithm was able to find a situation in the program's initial state which lead to the non-termination of the program. The description of this situation includes the description of the structure of the heap. The counter example that was found is actually the simplest possible one. This is of course not the most general description of a critical situation, but a very useful one because it explains the problems in the program's execution without overwhelming complex details.

8.4.5 Example Takeshi

The last example[12] is a more tricky one. It is called Takeshi and its code is shown in Figure 8.5. It uses the same linked list data structure as the example Traverse. The definition of the class Node is shown in Figure 8.4 in the description of the Traverse example.

The program Takeshi traverses the linked list starting with the head the same way as it does in example Traverse. But this time it appends a new node at the end of the list in each iteration of the loop. That means the list grows as fast as it is traversed, provided that tail follows head in the same list. Thus the traversal never reaches the end of the list. This is a very tricky example because the form of the heap is changed during the execution of the loop and the program itself does not know if the input nodes head and tail are actually connected in the list.

Preparations

The preparations of the program are the same as for the example Traverse, with the exception that we have to define the second input, the last node of the list, as well.

$$\begin{gathered}\Rightarrow \texttt{idHead} \geqslant 0 \land \texttt{idHead} < \texttt{kNodes} \land \texttt{idTail} \geqslant 0 \land \\ \texttt{idTail} < \texttt{kNodes} \land \{\ \ldots\ \}(\texttt{a}! = \texttt{null} \land \texttt{z}! = \texttt{null} \to \\ [\texttt{takeshi(a,z);}]\,\mathit{false})\end{gathered}$$

A modifier set of this program is more complex than for the Traverse example, because in the loop new objects of type Node are created. With each object creation

[12] We thank Mattias Ulbrich for the contribution of this example.

```
takeshi(Node head, Node tail) {
  Node run = head;
  while (run != null) {
    Node fresh = new Node(5);
    tail.succ = fresh;
    tail = tail.succ;
    run = run.succ;
  }
}
```

Figure 8.5: Takeshi

This is a program which traverses a linked list and adds a new node in at the end of the list with each step of the traversal. This program does not terminate whenever `run` and `tail` are elements of the same list and `run` is positioned before `tail` in the list. The definition of the class `Node` is already given in Figure 8.4.

the counter $\mathrm{next}_{\texttt{Node}}$ is increased. Additionally the attributes of the new `Node` object are assigned new values. A modifier set is then the following.

$$\{\texttt{run},\ \texttt{tail},\ \mathrm{next}_{\texttt{Node}}, \texttt{value}_{\texttt{Node}}, \texttt{succ}_{\texttt{Node}}\}$$

Application of the algorithm

Iteration No. 1 With the invariant candidate $inv_1 \equiv true$, the first iteration of the algorithm is performed. The result is an open proof with several open goals. The interesting formula in the antecedent of one of the goals is

$$\texttt{run} = \texttt{null}$$

It is the negation of the loop condition and thus we add the loop condition to the invariant candidate and obtain inv_2.

$$inv_2 \equiv true \wedge \texttt{run} \not\doteq \texttt{null}$$

Iteration no 2 The second iteration of the algorithm yields several open goals. We will not state them explicitly here because of space limitations. The formula which is the most promising one for our purpose is

$$\texttt{tail} = \texttt{run}$$

from the succedent of the open goal. It describes the situation when the `tail` and the `run` variable point to the same object. We add the formula to the invariant candidate and obtain inv_3.

$$inv_3 \equiv true \wedge \texttt{run} \not\doteq \texttt{null} \wedge \texttt{tail} = \texttt{run}$$

And with this invariant candiate the proof can be closed. The resulting constraint is the following[13].

$$(\texttt{idTail} = \texttt{idHead} \land -1 < \texttt{idHead} \land -1 < \texttt{idTail} \land \texttt{idTail} < \texttt{kNodes}) \lor \ldots$$

`kNodes` is the number of nodes in the list and `idHead` and `idTail` are the positions of the nodes `head` and `tail` in the list. The constraint basically says that the nodes `head` and `tail` are identical and within range.

Evaluation

The algorithm solved this example by finding a trivial counterexample where `run` and `tail` point to the same object. Interestingly, no human interaction or extraordinary invariants were necessary. We simply followed the description of the algorithm in Chapter 5 using only the creation methods no 1 and 2.

This example was the trickiest one we found so far because it uses a heap structure whose appearance is changed during the loop execution and yet the algorithm was powerful enough to at least describe one situation where the program turns into an endless loop, even if it was not the most general one.

8.4.6 Evaluation of the Algorithm for Heap Programs

We presented four examples of non-terminating Heap programs. The examples contained common datastructures which make use of the heap, namely arrays and linked lists. They contained programming errors which could realistically occur in the software development process.

The experiments showed that the algorithm works on such programs in principle. It solved the examples without any modifications in its behavior. Only the example Traverse needed some slightly enhanced initial invariant candidate, whose necessity we justified in Section 8.4.4. This candidate could be created automatically in an actual deployment of the software in future.

In conclusion, the experiments yielded good results which lead to our assumption that the approach is worth further exploring concerning the application to Heap programs.

[13] We simplified the constraint to make it more human readable.

Chapter 9

Summary and Conclusion

Although the idea of termination analysis of programs was raised decades ago, no one actually tried to develop a tool which specifically looks for non-termination in imperative programs. We made the first step into this direction and there are many more to go. Because this part of termination analysis is only rarely investigated so far, there was no standardized problem set which could be used to compare the quality of different approaches. As a starting point, we built up a database of non-terminating programs to be able to compare different approaches and settings.

We developed an algorithm to analyze programs for non-termination. In the development, we examined how to express the non-termination of a program in dynamic logic. The starting point of the algorithm is a formula which existentially quantifies over all possible program inputs and states the non-termination of the program. In the execution of the algorithm, invariants are generated. These invariants are used to prove the non-termination of the program with the help of an external theorem prover. The output of a successful run of the algorithm is a description of the set of inputs which lead the program into an infinite loop.

We implemented the algorithm as a Java software using the theorem prover KeY as back end. The software provides heuristics for the search of the necessary invariant. We ran a number of experiments on our sample database to explore the usefulness of the heuristics and measure the quality of our approach.

Our software is able to prove the non-termination of 75% of the examples in the database. In many cases the tool found a description of the set of critical inputs which was as general as possible. For other problems at least a counter example for the termination of the program was found, sometimes even the simplest one. Among the solved problems were all programs which we considered as common programming errors.

Our tool can be seen as the prototype of a debugging tool, alerting the programmer when she is about to write a non-terminating program. Our approach is thus an enrichment to the software development process. It attacks the problem of termination analysis from the other side, namely the non-termination side. It can thus be considered as a complement to algorithms which analyze programs with focus on termination only. There is the idea of running two analysis tools at the same time: one focussing on the termination, one on non-termination. The tool which terminates sucessfully at first announces the result of the overall analysis then.

9.1 Related Work

In Chapter 1, we already mentioned research projects in the field of non-termination. Now that we have presented our approach, we would like to additionally mention some studies, which use similar methods as ours.

In our approach we generate invariants which are in particular useful for the proof of non-termination. The field of invariant generation in general is heavily researched though. Most of the projects here are interested in generating invariants by examining the relations between program variables. Those invariants are generated without the particular purpose of proving non-termination. Most of these projects are based on the work of Cousot in the 70ies ([CH78]), which introduced and used abstract interpretation. Recent publications on this topic are [Lei05], [BL99], [BBM97] and [RCK04] for example.

[BBM97] compares the two basic approaches to prove that a given property does hold in a program's loop execution starting from a specified set of start states: forward propagation and backward propagation.

Forward propagation starts with the given set of start states as initial set. Then, the operation which is defined in the loop body is performed on the states of the set. The states which are reached by this operation are included into the set of states by widening of the property which describes the (initial) set. This step is performed until no new states can be included anymore. In other words, a fixpoint iteration on the set of states is performed until a fixpoint is reached. If the set which is the fixpoint is a subset of the set of states which fulfill the property in question, then we have proven that the property is preserved in the loop execution.

In contrast, backward propagation starts with the set of states for which the property in question holds. Then we examine if there are states which are reachable by execution of the loop body from states where the property does not hold. These states, which are reachable from states which violate the property, are then excluded from the set of states. We perform this operation on the set of states until the set is not reduced anymore. This is also a fixpoint operation on the set of states. The resulting set of states contains the states from which the program can be started so that the property in question is preserved. If the given set of start states is a subset of this one, we have proven that the program preserves the property for the given set of start states.

Our algorithm has the characteristics of backward propagation, although in our case the property in question, namely the invariant, is not given beforehand, but refined in the process of backward propagation. We start with the most general set of states which is described by the invariant candidate *true*. The refinement of the invariant candidate by formulae from open goals of the third branch, the use-case branch of the invariant rule, makes sure that the invariant candidate implies the loop condition. This leads to a description of a useful set of start states for an infinite loop[1].

Then the invariant candidate is further refined by open goals from the second branch, the body-branch of the invariant rule. This refinement is similar to the refinement in backward propagation, because the dynamic-logic formula describing that the invariant is preserved is equivalent to the formula of the weakest precondition which is used to define the fixpoint iteration of the backward propagation in [BBM97]. The difference of our refinement methods to the one of [BBM97] is that we sometimes refine too much and actually exclude states from the invariant although none of their predecessors violate the invariant.

This is the reason for that we sometimes end up with an invariant candidate which prevents the first branch, the init-branch of the invariant rule, from closing. In this case the invariant candidate describes a set of states from which if the program is started in them, it can never reach a state from which the loop is executed. The check if the first branch can be closed[2] then is actually the check if the refinement was too coarse. In this case we have to backtrack and try a different invariant candidate. This cannot happen in the classical way of backward propagation, because here no states are removed from the invariant that are reached by states that violate the invariant.

We suggested in Section 7.7 the combination of our method and other methods of invariant generation (for example by abstract interpretation). In experiments, we used an invariant generator which was developed by Weiß [Wei07] in the scope of the KEY PROJECT. This generator is based on abstract interpretation and makes use of the KEY prover which was also the basis of our implementation. We used this invariant generator to produce invariants which were intended to serve as initial invariant in our algorithm (instead of the formula *true*). Unfortunately, those invariants were too general to contribute any further information than the formulae *true* or the loop condition. Thus these invariants did not much improve the process of non-termination invariant generation. Nevertheless we think that this idea is worth further investigation, because we could only try one invariant generator on a limited number of examples.

Disproving the termination of programs is only one way to prove the incorrectness of programs. Our algorithm produces a description of program inputs which make the program loop endlessly. We thus generate counter examples for the termination of programs. Rümmer uses a similar approach to generate counter examples for program correctness in other aspects. His approach also uses formulae which

[1] The start state is useful, because if the loop condition was not fulfilled and thus the loop never executed, it would not make sense to look for an infinite loop.

[2] This is filter method no 3 in Section 5.2.2.

existentially quantify over program inputs as starting point and a constraint solver to find a description of the critical input states. For details see [RS07] and [Rüm07].

As we mentioned in Chapter 1, there are research groups which develop tools to automatically prove the termination of a program. Termination proofs are done by finding a ranking term or function, which maps the states which the program traverses during the loop execution into a well-founded domain and showing that the value of the term decreases in each iteration of the loop. Because in well-founded domains no infinite descending chains exist, the term finally must reach a minimum element and thus the loop terminates.

We let us inspire by this technique and discussed the idea of inverse ranking terms in Section 4.3 to prove non-termination. Like ranking terms, those inverse ranking terms are also mapped into an ordered domain. For proving the non-termination we simply prove that the term increases or stays stationary in each loop iteration instead of that it decreases. Assuming that the loop terminates if the term reaches a particular minimum element, we thus prove that the term "moves away" from the termination of the loop. Inverse ranking terms are a reasonable method do prove non-termination, but we could show in its discussion, that this approach is actually subsumed by the idea of non-termination invariants.

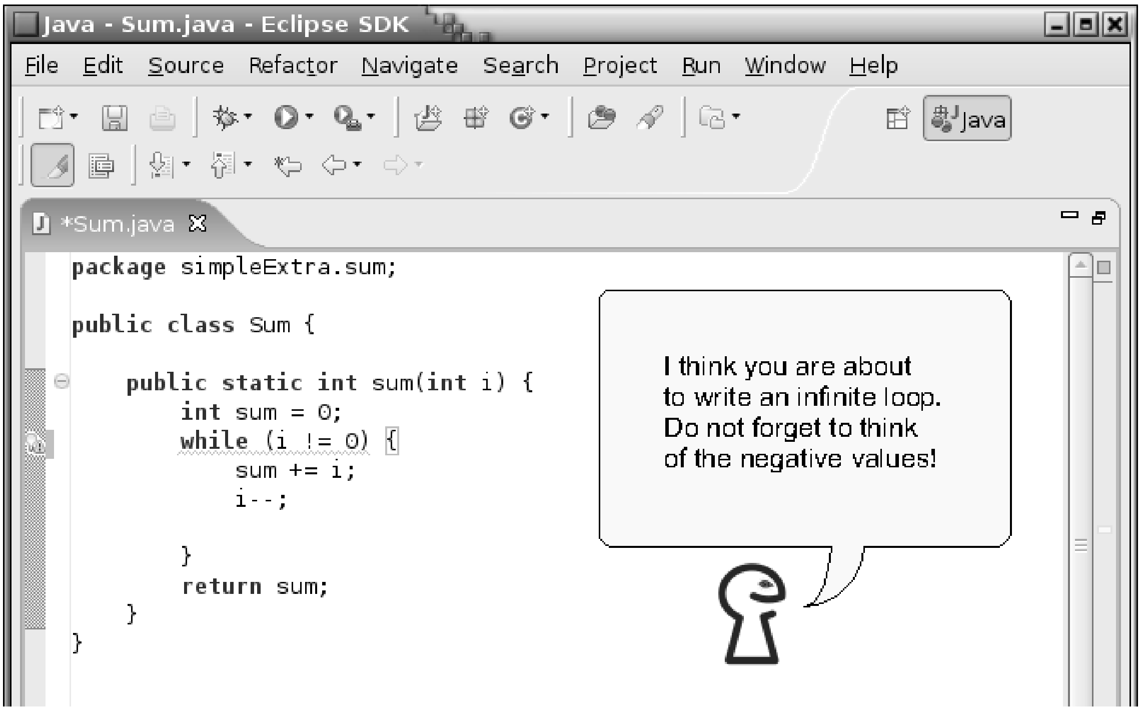
The software we developed in the scope of this thesis is based on the KeY prover. The prover is developed in the Key Project [BHS07], which is a joint project of the three European universities Chalmers Technical University in Gothenburg (Sweden), University of Karlsruhe (Germany) and University of Koblenz-Landau (Germany). The KeY prover is a software to verify that programs comply to their specification. Specifications do usually not only include the properties of termination of a program but also state more particular properties of the resulting state of a program.

9.2 Future Work

Our work is a proof of concept and as such gives us plenty of paths to go in the future. We examined and enumerated a number of ways to improve the algorithm itself to raise the quality of the results in Section 7.7. The most promising points of attack here are the improvement of the heuristics and the constraint solver.

Besides the mere purpose of non-termination invariant generation, there are further applications for which our approach might be useful. The idea of the algorithm to repeatedly produce proof (attempts) in order to refine an entity which is necessary for the proof itself can be applied to other situations as well. First of all, we are thinking of invariants for partial correctness proofs here.

In Chapter 8 we already had a peek into to the application of our algorithm on programs of richer programming languages, in our case Heap programs. It is a desirable situation to be able to find non-termination bugs in programs of any modern programming language which implements the concepts of object-orientation. We made the first step here, there are for sure a lot more to go, for example, analyzing the termination of programs with even more complex data structures than the ones we examined in this work.



Java - Sum.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
*Sum.java

```
package simpleExtra.sum;

public class Sum {

    public static int sum(int i) {
        int sum = 0;
        while (i != 0) {
            sum += i;
            i--;

        }
        return sum;
    }
}
```

I think you are about
to write an infinite loop.
Do not forget to think
of the negative values!

Figure 9.1: A Development Environment in the Future

It is a major goal of the formal verification community to integrate the methods of verification into the every-day software development process. The more convenient the application of those methods of formal verification is, the bigger is their impact on the quality of software. We have the vision of a non-termination checker which is integrated into modern software development environments (Figure 9.1). One day, warnings will pop up the minute we write the code of an endless loop. However, no matter how well we try to achieve this goal, we will never solve the halting problem. So, may Alan Turing rest in peace.

Appendix A

While Programs Database

The following programs where already shown in the thesis. See the respective figures: AlternatingIncr (Figure 4.4), AlternDivWidening (Figure 5.8), AlternKonv (Figure 7.8), Collatz (Figure 3.1), Ex02 (Figure 7.7), Factorial (Figure 7.10), Gauss (Figure 2.2), UpAndDown (Figure 4.1), and WhileNestedOffset (Figure 5.11).

```
alternDiv(int i) {
  while (i != 0) {
    if (i < 0) {
      i--;
      i = i*(-1);
    } else {
      i++;
      i = i*(-1);
    }
  }
}
```

Figure A.1: AlternDiv

```
alternDivWide(int i) {
  int w = 5;
  while (i != 0) {
    if (i < -w) {
      i--;
      i = i*(-1);
    } else {
      if (i > w) {
        i++;
        i = i*(-1);
      } else {
        i = 0;
      }
    }
  }
}
```

Figure A.2: AlternDivWide

```
complInterv(int i) {
  while (i*i > 9) {
    if (i < 0) {
      i = i-1;
    } else {
      i = i+1;
    }
  }
}
```

Figure A.3: ComplInterv

```
complInterv2(int i) {
  while (i != 0) {
    if (i > -5 && i < 5) {
      if (i < 0) {
        i++;
      }
      if (i > 0) {
        i--;
      }
    }
  }
}
```

Figure A.4: ComplInterv2

```
complInterv3(int i) {
  while (i != 0) {
    if (i > 5) {
      i++;
    } else {
      if (i < -5) {
        i--;
      } else {
        i = 0;
      }
    }
  }
}
```

Figure A.5: ComplInterv3

```
complxStruc(int i) {
  int j = i;
  while (i > 0) {
    if (i >= j) {
      i--;
      if (j < 5) {
        j++;
        if (i-j>2) {
          i++;
        } else {
          j++;
        }
      } else {
        j--;
      }
    } else {
      if (i > 0 & j < 0) {
        i--;
        if (j < -1) {
          j++;
        } else {
          i++;
        }
      } else {
        i++;
        if (j*2 > i) {
          j--;
        } else {
          j++;
        }
      }
    }
  }
}
```

Figure A.6: ComplxStruc

```
convLower(int i) {
  while (i > 5) {
    if (i != 10) {
      i--;
    }
  }
}
```

Figure A.7: ConvLower

```
cousot(int i, int j) {
  while (true) {
    if (i < j) {
      i = i+4;
    } else {
      j = j+1;
      i = i+2;
    }
  }
}
```

Figure A.8: Cousot

```
doubleNeg(int i, int j) {
  while (i*j > 0) {
    i--;
    j--;
  }
}
```

Figure A.9: DoubleNeg

```
even(int i) {
  while (i != 1 && i != 0) {
    i = i-2;
  }
  return (i == 0);
}
```

Figure A.10: Even

```
ex01(int i) {
  while (i < 0) {
    i--;
  }
}
```

Figure A.11: Ex01

```
ex03(int i) {
  while (i < 0) {
    if (i != -5) {
      i++;
    }
  }
}
```

Figure A.12: Ex03

```
ex04(int i) {
  while (true) {
    i--;
  }
}
```

Figure A.13: Ex04

```
ex05(int i) {
  while (true) {
    ;
  }
}
```

Figure A.14: Ex05

```
ex06(int i) {
  while (i >= -5 && i <= 5) {
    if (i > 0) {
      i--;
    }
    if (i < 0) {
      i++;
    }
  }
}
```

Figure A.15: Ex06

```
ex07(int i) {
  while (true) {
    if (i > 0) {
      i--;
    }
    if (i < 0) {
      i++;
    }
  }
}
```

Figure A.16: Ex07

```
ex08(int i) {
  boolean up = false;
  while (i > 0) {
    if (i == 1) {
      up = true;
    }
    if (i == 10) {
      up = false;
    }
    if (up) {
      i++;
    } else {
      i--;
    }
  }
}
```

Figure A.17: Ex08

```
ex09half(int i) {
  int l = i;
  i = 0;
  while (l - i > 0) {
    i = i + (l - i) / 2;
  }
}
```

Figure A.18: Ex09Half

```
fib(int n) {
  int i = 0;
  int j = 1;
  int t = 0;
  while (j != n) {
    t = j+i;
    i = j;
    j = t;
  }
}
```

Figure A.19: Fib

```
flip(int i, int j) {
  int t = 0;
  while (i != 0 && j != 0) {
    t = i;
    i = j;
    j = t;
  }
}
```

Figure A.20: Flip

```
flip(int i, int j) {
  int t = 0;
  while (i > 0 && j > 0) {
    if (i < j) {
      t = i;
      i = j;
      j = t;
    } else {
      if (i > j) {
        j = i;
      } else {
        i--;
      }
    }
  }
}
```

Figure A.21: Flip2

```
gcd(int a, int b) {
  int t = 0;
  if (b > a) {
    t = a;
    a = b;
    b = t;
  }
  while (b != 0) {
    t = a-b;
    a = b;
    b = t;
  }
  return a;
}
```

Figure A.22: Gcd

```
lcm(int a, int b) {
  int am = a;
  int bm = b;
  while (am != bm) {
    if (am > bm) {
      bm = bm+b;
    } else {
      am = am+a;
    }
  }
  return am;
}
```

Figure A.23: Lcm

```
marbie1(int i) {
  while (i > 2) {
    i++;
  }
}
```

Figure A.24: Marbie1

```
marbie2(int i) {
  while(5<8) {
    i++;
  }
}
```

Figure A.25: Marbie2

```
middle(int i, int j) {
  while (i != j) {
    i--;
    j++;
  }
  return i;
}
```

Figure A.26: Middle

```
mirrorInterv(int i) {
  int range = 20;
  while (-range <= i & i <= range) {
    if (range-i < 5 || range+i < 5) {
      i = i*(-1);
    } else {
      range++;
      i--;
      if (i == 0) {
        range = -1;
      }
    }
  }
}
```

Figure A.27: MirrorInterv

```
mirrorIntervSim(int i) {
  while (i != 0) {
    if (-5 <= i && i <= 35) {
      if (i < 0) {
        i = -5;
      } else {
        if (i > 30) {
          i = 35;
        } else {
          i--;
        }
      }
    } else {
      i = 0;
    }
  }
}
```

Figure A.28: MirrorIntervSim

```
moduloLower(int n) {
  while (n > 2) {
    if (n % 5 > 0) {
      n--;
    }
  }
}
```

Figure A.29: ModuloLower

```
moduloUp(int n) {
  int d = 10;
  while (n < 15) {
    n++;
    n = n % d;
  }
}
```

Figure A.30: ModuloUp

```
narrowing(int i) {
  int range = 20;
  boolean up = false;
  while (0 <= i && i <= range) {
    if (i == 0) {
      up = true;
    }
    if (i == range) {
      up = false;
    }
    if (up) {
      i++;
    }
    if (!up) {
      i--;
    }
    if (i == range-2) {
      range--;
    }
  }
}
```

Figure A.31: Narrowing

```
narrowKonv(int i) {
  int range = 20;
  while (0 <= i && i <= range) {
    if (!(0 == i && i == range)) {
      if (i == range) {
        i = 0;
        range--;
      } else {
        i++;
      }
    }
  }
}
```

Figure A.32: NarrowKonv

```
plait(int i, int j, int k) {
  int plaitNext = 0;
  int swap = 0;
  while (i > 0 || j > 0 || k > 0) {
    if (plaitNext == 0) {
      swap = i;
      i = j/2;
      j = swap*2;
      plaitNext = 1;
    } else {
      swap = k;
      k = j*2;
      j = swap/2;
      plaitNext = 0;
    }
  }
}
```

Figure A.33: Plait

```
sunset(int i) {
  while (i > 10) {
    if (i == 25) {
      i = 30;
    }
    if (i <= 30) {
      i--;
    } else {
      i = 20;
    }
  }
}
```

Figure A.34: Sunset

```
trueDiv(int i) {
  while (true) {
    if (i <= 0) {
      i--;
    } else {
      i++;
    }
  }
}
```

Figure A.35: TrueDiv

```
twoFloatInterv(int i) {
  while (i > 0 & i < 50) {
    if (i < 20) {
      i--;
    }
    if (i > 10) {
      i++;
    }
    if (30 <= i && i <= 40) {
      i--;
    }
  }
}
```

Figure A.36: TwoFloatInterv

```
upAndDownIneq(int i) {
  int up = 0;
  while (0 <= i && i <= 10) {
    if (i >= 10) {
      up = 0;
    }
    if (i <= 0) {
      up = 1;
    }
    if (up >= 1) {
      i++;
    } else {
      i--;
    }
  }
}
```

Figure A.37: UpAndDownIneq

```
whileBreak(int i) {
  while (i > 10) {
    if (i > 20) {
      i++;
    } else {
      i--;
    }
    if (i == 30) {
      break;
    }
  }
}
```

Figure A.38: WhileBreak

```
whileDecr(int i) {
  while (i > 5) {
    i--;
  }
}
```

Figure A.39: WhileDecr

```
whileIncr(int i) {
  while (i > 0) {
    i++;
  }
}
```

Figure A.40: WhileIncr

```
whileIncrPart(int i) {
  while (i > 0) {
    if (i > 3) {
      i++;
    } else {
      i--;
    }
  }
}
```

Figure A.41: WhileIncrPart

```
whileNested(int i) {
  int j;
  while (i < 10) {
    j = i;
    while (j > 0) {
      j++;
    }
    i++;
  }
}
```

Figure A.42: WhileNested

```
whilePart(int i) {
  while (i > 5) {
    if (i < 10) {
      i--;
    }
  }
}
```

Figure A.43: WhilePart

```
whileSingle(int i) {
  while (i < 10) {
    if (i != 3) {
      i++;
    }
  }
}
```

Figure A.44: WhileSingle

```
whileSum(int i, int j) {
  while (i+j > 0) {
    i++;
    if (j % 2 == 0) {
      j = j - 2;
    }
  }
}
```

Figure A.45: WhileSum

```
whileTrue(int i) {
  while (true) {
    i++;
  }
}
```

Figure A.46: WhileTrue

List of Figures

Bibliography

[AZ95] T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *In Proceedings of the Fifth Workshop on Logic Program Synthesis and Transformation*, LNCS, pages 219–233. Springer–Verlag, 1995.

[BBM97] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[BL99] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15(1):75–92, 1999.

[Bou92] R. T. Boute. The Euclidean definition of the functions div and mod. *ACM Transactions on Programming Languages and Systems*, 14(2):127–144, 1992.

[BP06] B. Beckert and A. Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, LNCS 4130, pages 266–280. Springer, 2006.

[BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.

[BS04] B. Beckert and S. Schlager. Software verification with integrated data type refinement for integer arithmetic. In *Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK*, LNCS 2999, pages 207–226. Springer, 2004.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Program-*

ming Languages, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[Che00] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CPR] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV '06, Proceedings of the 18th International Conference on Computer-Aided Verification*, pages 415–418.

[CS01] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 67–81, London, UK, 2001. Springer-Verlag.

[CS02] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 442–454, London, UK, 2002. Springer-Verlag.

[Fit96] M. Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Gie01] M. Giese. Incremental closure of free variable tableaux. In *IJCAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 545–560, London, UK, 2001. Springer-Verlag.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2005.

[Göd31] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[GSKT05] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Proving and disproving termination of higher-order functions. In *FroCoS '05: Proceedings of the 5th International Workshop on Frontiers of Combining Systems, Vienna, Austria*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231, 2005.

[GSKT06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *IJCAR '06: Proceedings of the 3rd International Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 281–286, London, UK, 2006. Springer-Verlag.

[HKT00] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* The MIT Press, Cambridge, Massachusetts, London, 2000.

[HPRW06] R. Hähnle, J. Pan, P. Rümmer, and D. Walter. Integration of a security type system into a program logic. In *TGC '06: Proceedings 2nd Symposium on Trustworthy Global Computing, Lucca, Italy*, LNCS, London, UK, 2006. Springer-Verlag.

[Käu05] C. Käunicke. Automatic termination analysis of logic programs, 2005. Diploma Thesis, RWTH Aachen, Germany.

[Kin76] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[Lei05] K. Rustan M. Leino. Invariants on demand. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 148–149, Washington, DC, USA, 2005. IEEE Computer Society.

[MZ07] C. Marché and H. Zantema. The termination competition 2007. 2007. `http://www.lri.fr/~marche/termination-competition/`.

[Pre91] M. Pressburger. *On the completeness of a certain sysetm of arithmetic of whole numbers in which addition occurs as the only operation (reprint)*, volume 12. Hist. Philos. Logic, 1991.

[PSS97] S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically proving termination of programs in a non-strict higher-order functional language. In *Proceedings of the 4th International Static Analysis Symposium, Paris*, LNCS 1302, pages 345–360, 1997.

[RCK04] 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.

[RS07] P. Rümmer and M. A. Shah. Proving programs incorrect using a sequent calculus for Java Dynamic Logic. In *International Conference on Tests And Proofs (TAP)*, London, UK, 2007. Springer-Verlag. To appear.

[Rüm06] P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Logic for Programming, Artificial Intelligence and Reasoning*, volume 4246, pages 422–436, London, UK, 2006. Springer-Verlag.

[Rüm07] P. Rümmer. A sequent calculus for integer arithmetic with counterexample generation. In *VERIFY'07: 4th International Verification Workshop at CADE 21, Bremen, Germany*, 2007. `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-259/`.

[Rüm08] P. Rümmer. Notes on constraints in non-destructive first-order calculi. 2008. To appear.

[Son06] M. Sondermann. Automatische Terminierungsanalyse von imperativen Programmen, 2006. Diploma Thesis, RWTH Aachen, Germany.

[Swi05] S. Swiderski. Terminierungsanalyse von Haskellprogrammen, 2005. Diploma Thesis, RWTH Aachen, Germany.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Wei07] B. Weiß. Inferring invariants by static analysis in KeY, 2007. Diploma Thesis, University of Karlsruhe, Germany.