RWTH Aachen

Department of Computer Science
Technical Report

Java Program Analysis by Symbolic Execution

Carsten Otto

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2013-16

RWTH Aachen · Department of Computer Science · März 2015

Java Program Analysis by Symbolic Execution

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Carsten Otto
aus
Tönisvorst

Berichter: Univ.-Prof. Dr. Jürgen Giesl
Univ.-Prof. Fausto Spoto, PhD

Tag der mündlichen Prüfung: 03. März 2015

Carsten Otto
Lehr- und Forschungsgebiet Informatik 2
otto@informatik.rwth-aachen.de

Aachener Informatik-Bericht AIB-2013-16

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Program analysis has a long history in computer science. Even when only considering the important aspect of termination analysis, in the past decades an overwhelming number of different techniques has been developed. While the programming languages considered by these approaches initially were more of theoretical importance than of practical use, recently also automated analyses for imperative programming languages like C or Java have been developed. Here, a major challenge is to deal with language constructs and concepts which do not exist in simpler languages. For example, in Java one often uses dynamic dispatch, complex object hierarchies, or side-effects with far-reaching consequences involving the global heap.

In this thesis, we present a preprocessing step for Java Bytecode programs in which all such complicated language constructs are handled. This way, subsequent analyses do not need to be concerned with these, and making use of existing techniques is easy. In particular, we show how Symbolic Execution Graphs can be constructed which contain an over-approximation of all possible program runs. This way, and by taking care of having a precise approximation, the information contained in the constructed graphs can, for example, be used to reason about the termination behavior of the original program.

Additionally to the construction of such graphs, in this thesis we present a new analysis technique which helps end users identify parts of the analyzed code which are irrelevant for the desired outcome. This way, programming errors causing code to be not executed can be identified and, consequently, fixed by the user. For this technique to be useful, the information contained in the previously constructed graph needs to be precise. We will demonstrate that this is the case.

For the techniques presented in this thesis, a rigorous formalization is shown. To comply with the overall goal of, for example, *automated* termination analysis, we also need to implement the techniques and theoretical results. In this thesis we show how certain hard to automate aspects can be approached, leading to a competitive implementation.

The techniques presented in this thesis are implemented in the AProVE tool. As also related techniques working on Symbolic Execution Graphs are implemented in AProVE, with the click of a button users can analyze Java Bytecode programs for (non)termination and find irrelevant code. In the annual International Termination Competition, it is demonstrated that currently AProVE is the most powerful termination analyzer for Java Bytecode programs.

Acknowledgments

First of all, I want to thank Jürgen Giesl for being my first supervisor, and more importantly also for his guidance during my time in his research group. I enjoyed a lot of freedom, his constant support, and Jürgen was always available to discuss and tackle arising challenges.

I also want to thank Fausto Spoto, who agreed to be my second supervisor. I am grateful for his creation of `Julia`, as good competition makes research more interesting.

Marc Brockschmidt has been a great office mate in the past few years, and I am very thankful for our detailed discussions and exchanges of mind-boggling ideas. Not only Marc, but also my other colleagues Fabian Emmes, Martin Plücker, Thomas Ströder, and Stephanie Swiderski always made me look forward to the next day in the office, and offered both reasonable and irrational ways to deal with the ups and downs of research, teaching, life, and everything. In this context I also want to thank all colleagues and students on the same floor, who helped in creating a relaxed and pleasant atmosphere.

Thomas Noll and Marc proof-read parts of my thesis, for which I am very thankful. I also want to thank the countless developers of the free open source software which I was glad to use. Without them, even the simplest task would have been both expensive and frustrating.

Sincere thanks go to my family, especially my parents, who supported me my whole life. I should have called and visited more often, and I really hope this thesis at least compensates part of my neglect.

Last, but definitely not least, I want to thank Anke. While most of my work still is a mystery to her, she always met me with patience, support, motivation, and love. Without her, I would be a lonely guy, and most likely I would still be just about to start writing this thesis.

Carsten

Contents

Introduction

Software development is a very complex task. First, one needs to understand the requirements imposed by the client. Then, based on these requirements a software solution needs to be developed. Depending on the application, this usually means that the new work has to be integrated into existing systems. Furthermore, development should be fast. The resulting product should be designed such that other developers can easily extend and improve it. Last but not least, the software should always compute the correct results.

In this thesis we are concerned with the *correctness* of programs. A part of theoretical computer science is *software verification* where techniques are developed which help in analyzing whether a given program indeed is correct. One important aspect of any program is its termination behavior. A program usually is only considered correct if it never provides a wrong result and, in addition, always terminates after a finite time.

The results presented in this thesis are motivated by the question of computing the termination behavior of a given program. While this is undecidable in general [Tur36], in the past few decades many results have been published which help in analyzing the termination behavior of programs written in many different programming paradigms and languages. One of many examples is the analysis of *term rewrite systems* (TRSs). Work on termination analysis of TRSs has started as early as 1970 [MN70], and many additional results have been achieved since then. More recently, *transformational* techniques have been developed which transform programs of different paradigms into TRSs. Then, by making use of and by extending existing techniques analyzing TRSs, termination of such programs can be shown. While in [Sch08, SGST09, SGS+10, GRS+11, GSS+12] such transformational approaches have been developed for declarative programming languages, in this thesis we show a related approach for the imperative programming language Java.

Common for all of these techniques is the idea to deal with all language-specific problems by creating a graph representing all computations of the program. Based on this graph then TRSs are created where termination of the TRSs implies termination of the original program. With this approach concepts like *unification* (in the case of logic programming) and *lazy evaluation* (in the case of functional programming languages), which are hard to formalize using term rewriting, can be handled during graph construction. Likewise, in this thesis we handle concepts like *aliasing* and *dynamic dispatch* which often occur in imperative programs during graph construction.

Contributions

The author of this thesis co-authored five peer-reviewed papers related to the analysis of Java Bytecode programs [OBvEG10, BOvEG10, BOG11, BSOG12, BMOG12]. In this thesis, several techniques presented in [OBvEG10, BOvEG10, BOG11] are combined and extended substantially.

The results published in [OBvEG10] were developed together with the co-authors Marc Brockschmidt and Christian von Essen as part of their Diploma theses. The author of the current thesis contributed the key ideas and fundamental design choices, and together with the co-authors worked on refining them. For [BOvEG10] the author of this thesis refactored and extended the implementation, in parallel to the development of the formalization presented in the publication. The ideas leading to the results published in [BOG11] were implemented by the author of this thesis. Moreover, the author of this thesis substantially extended the approach in [BOG11] to a prototypical implementation which also works with heap predicates. This is a crucial and highly challenging extension that is not part of [BOG11].

In addition, a new and unpublished technique, developed by the author of this thesis, building on the results presented in these papers is presented. Using this new technique it is possible to identify specific kinds of bugs automatically, which otherwise are hard to find. Furthermore, we give algorithms corresponding to the presented techniques, whose automation is not trivial.

Symbolic Execution Graphs for Non-Recursive Programs

In Chapter 1 we present a technique which transforms Java Bytecode programs into Symbolic Execution Graphs. These graphs then can be used, for example, for termination analysis of Java Bytecode programs. In [OBvEG10] we presented the main ideas of this approach.

In this thesis, we describe a more complete approach which considers aspects of Java Bytecode which were left out of the paper. This includes static fields, arrays, exceptions, class initialization, and return addresses. A key part of the state representation, namely how to represent fields for object instances, is refined so that the formalization as presented in this thesis allows for a more precise analysis. Furthermore, the power of some annotations (named heap predicates in this thesis) is strengthened. In Section 1.5 we present *state intersection* which makes it possible to intersect the information represented in two abstract states. Using this technique we can obtain more precise information when using *equality refinement*, where in [OBvEG10] only a rather trivial approach was presented. In [BOvEG10] we have shown that the symbolic execution as described in [OBvEG10] indeed is correct. While certain parts of a corresponding proof are not shown in [BOvEG10], in this thesis we show a complete proof regarding correctness of how the

PUTFIELD opcode is evaluated. In this proof we consider all aspects of Java Bytecode, i.e., static fields etc. as described above. Additionally, mistakes in [OBvEG10, BOvEG10] w.r.t. PUTFIELD are fixed. Furthermore, as the handling of arrays differs from the handling of object instances in an important aspect, in this thesis we also show correctness of the opcodes AASTORE and AALOAD. With help of this formalization further bugs in the implementation were found. Finally, we show that the abstraction of [OBvEG10] has a finite depth, i.e., the analysis indeed can be used to create *finite* Symbolic Execution Graphs with the desired properties. In addition to this theoretical result, we show an algorithm used to create Symbolic Execution Graphs.

Automation

The implementation of the techniques presented in this thesis and the aforementioned papers is rather involved. In Chapter 2 we present corresponding algorithms and discuss how certain non-trivial aspects can be automated.

Recursion

In [BOG11] we presented an extension to the approach of [OBvEG10] so that also *recursive* programs can be analyzed. However, in [BOG11], we decided not to allow usage of annotations (resp. heap predicates) in the states. In Chapter 3, we extend the main concept of *context concretization* that is needed to handle recursive programs to also work on states containing heap predicates. As this extension was more complicated than anticipated, not all results presented in [BOG11] could be lifted to the setting of this thesis. Most importantly, the graph construction may not terminate even when using the key concepts of [BOG11]. In Section 3.6 a detailed discussion of this problem and possible solutions is given.

Bug Detection

In [BSOG12] we extended the analysis of [OBvEG10, BOG11] to find bugs related to NullPointerExceptions and non-termination in a given Java Bytecode program. In Chapter 4 we present another, unpublished analysis. As a result of this analysis, the user is shown parts of the program which are proven to be irrelevant, based on a definition of relevant results given by the user. Here, we make use of the detailed information contained in a Symbolic Execution Graph and use this information to reason about how information propagates in the program.

Implementation

The results of [OBvEG10, BOvEG10, BOG11, BSOG12, BMOG12] and most results of this thesis are implemented in the tool AProVE [GBE+14]. Of all the work which finally cumulated in this thesis, the author devoted a significant part to the implementation and related software engineering issues. In the time in which the author was actively involved in the development of the AProVE tool, the source code base of the whole project grew from about 240,000 source lines of code (SLoC) to about 550,000 SLoC. Of those, about 52,000 SLoC are directly related to the analysis of Java Bytecode.

As demonstrated in the annual termination competition[1], this implementation indeed is very powerful. In all competitions from 2009 to 2014, AProVE was the most powerful tool in the category for non-recursive Java Bytecode programs. In addition, starting with the ideas first published in [BOG11] and extended in Chapter 3, AProVE was the most powerful tool in the category for recursive JBC programs in all competitions from 2011 to 2014. For a more detailed discussion we refer to [Bro14].

New Contributions

They main concepts of Chapters 1 and 3 have been published in [OBvEG10, BOvEG10, BOG11]. Furthermore, this thesis contains ideas first published in [BSOG12, BMOG12, Bro10].

In the PhD thesis of Marc Brockschmidt [Bro14] ideas related to the current thesis are also discussed. Only by combining the disjoint contributions of both the current thesis and the contributions of [Bro14] it is possible to analyze the termination behavior of Java Bytecode programs. While the current thesis concentrates on the construction of Symbolic Execution Graphs as a *frontend* of the analysis, in [Bro14] the corresponding *backend* is shown. In particular, in [Bro14] it is shown how Symbolic Execution Graphs can be transformed into a variant of term rewrite systems, and how the termination behavior of such TRSs can be shown. Also note that the information contained in Symbolic Execution Graphs can also be used for further analyses, not just termination analysis as shown in [Bro14]. One such analysis is shown in Chapter 4.

The current thesis contains many improvements of the already published results and several novel contributions:

Chapter 1 In this thesis we extend the results first published in [Bro10, OBvEG10, BOvEG10, BOG11, BSOG12, BMOG12].

- We enhance the formalization of states by re-defining how fields of object instances are represented (Section 1.2). Because of this, less references and heap predicates are created when performing *realization refinement* (Section 1.4.5).

[1]See http://www.termination-portal.org/wiki/Termination-Competition

- Usage of the heap predicate $r =^? r'$ as introduced in [OBvEG10, Bro10] is limited such that at most for one of r, r', instance field information may be represented. In this thesis, we remove this restriction, enabling us to retain more precise information (Section 1.2.2).
- In this thesis we present a first formalization of *state intersection* (Section 1.5). While the concept of context concretization introduced in [BOG11] is strongly related, in [BOG11] the states may not contain heap predicates. Using state intersection, more precise results are obtained using *equality refinement* (Section 1.4.6).
- The correctness proof corresponding to the evaluation of PUTFIELD in abstract states was first shown in [BOvEG10]. In this thesis we extend the proof such that the enhanced definition of states as presented in this thesis is regarded (Section 1.6.1). Furthermore, work on this proof exposed a bug in the implementation of PUTFIELD (corresponding to a proof part omitted in [BOvEG10]).
- We first show correctness proofs for the opcodes AALOAD and AASTORE which work on arrays in abstract states (Sections 1.6.2 and 1.6.3). The state representation for object instances and arrays differs in an important aspect. Related proofs in the context of object instances were adapted accordingly, which also exposed bugs in the implementation.

Chapter 2 In [Bro10, BSOG12, BMOG12] variants of the *merge algorithm* are presented. However, in these algorithms operations on infinite sets are used, without giving further information on how to implement these operations. In Chapter 2 we give a detailed definition of the merge algorithm working only on finite sets.

Chapter 3 In [BOG11] no heap predicates were allowed, which is a severe limitation of the technique. In Chapter 3 we extend the main concept of *context concretization* to also work on states making use of heap predicates.

Chapter 4 All results of this chapter are unpublished.

Structure

After a short discussion of preliminaries, the contributions of this thesis are presented. In Chapter 1 we present Symbolic Execution Graphs and explain how these are constructed for non-recursive JAVA BYTECODE programs. This technique is based on symbolic execution and in the resulting graphs detailed information about the analyzed program is contained, which may be used for further analyses.

Then, in Chapter 2 we discuss how certain aspects of the graph construction can be automated. Here, we especially consider parts of the formalization involving infinite data structures and show corresponding finite structures which may be used for the implementation.

In order to also allow for an analysis of recursive programs, in Chapter 3 we discuss the problem of call stack abstraction. Then, we develop an extension to the analysis enabling this additional variant of abstraction.

Finally, in Chapter 4 we present an analysis which, based on the information provided in a Symbolic Execution Graph, finds irrelevant parts of the code. Based on these results, the user may discover bugs in the analyzed program. The thesis concludes with a discussion of the results and future work.

Preliminaries

Throughout this thesis we will work on programs written in Java Bytecode [LYBB12]. Java Bytecode is an assembly-like object-oriented language designed as intermediate format for the execution of Java [GJS+12] programs by a Java Virtual machine (JVM). Furthermore, Java Bytecode is the compilation target of other languages like Clojure [Hic08], Groovy [KGK+07], Scala [OSV08], Ruby (JRuby) [NES+11], Python (Jython) [PR02], and JavaScript (Rhino) [Rhi].

Java and Java Bytecode

Java is an imperative, object-oriented programming language that is used in many real world applications. Programs may work on data structures that reside on an unbounded heap. Because of side effects local changes may also be visible by other parts of the program. Using the concept of method overriding (dynamic dispatch) the target of a method invocation is determined at runtime. Furthermore, because of exceptions, finally blocks, and implicit code like static initializers, it is not easy to see which code is executed. As an example, invoking a static method might trigger execution of arbitrary code in the static initializer block of the corresponding class before the actual invocation is performed.

Since every Java program can automatically be compiled into a Java Bytecode program, Java Bytecode also contains most of the features of Java. In contrast to the Java syntax which is tailored to be user-friendly, the syntax of Java Bytecode is very limited. Because of this, Java Bytecode is the natural choice for automated analyzers which really are intended to analyze Java programs.

To ease presentation, in this thesis we will use examples written in Java although the presented techniques all work on Java Bytecode instead. This also means that, while at least basic knowledge of Java is required, a detailed understanding of Java Bytecode is not necessary to understand this thesis. However, a few basic concepts that distinguish Java Bytecode from Java need to be understood.

Program Format

While Java code is a human-readable string composed of pre-defined keywords, Java Bytecode is provided in a binary format. This binary format includes components for individual classes (similar to the case of JAVA), most commonly known as files ending in .class. For each class certain attributes (like the defined fields and details about the class hierarchy) are stored. In addition, the methods defined in the class are part of the representation. The code of each method is just a series of individual commands, so-called *opcodes*.

When evaluating any imperative program, in each computation step a state s is transformed into a state s' by changing certain parts of s based on the evaluated code. This also is the case for JAVA code. However, in JAVA code individual computation steps usually are combined into more complicated *statements*. For example, the statement

$$\mathsf{x} = \mathsf{f}() + \mathsf{b};$$

is a composition of four smaller steps:

(1) invoke the method f() and get the result

(2) get the content of the local variable b

(3) compute the addition

(4) store the result into the local variable x

In contrast to the wealth of syntactic sugar present in JAVA code, in JAVA BYTECODE there only exists a small number of opcodes which are combined in order to achieve the effects possible with complicated JAVA statements. For the example shown above, a corresponding opcode sequence could be

$$\mathsf{INVOKESTATIC}\ f, \mathsf{ILOAD\_0},\ \mathsf{IADD},\ \mathsf{ISTORE\_1}$$

which directly resembles the individual steps mentioned above.

Branches (and, therefore, also loops) in a JAVA program are realized in JAVA BYTECODE using (conditional) jumps. In the same way, exception handlers are just sequences of opcodes in the method for which the method contains the additional information where to jump to in case of certain exceptions.

To summarize, interpreting a JAVA BYTECODE program is performed by just applying the effects of a single (simple) opcode to the current state and then interpreting the subsequent opcodes, step by step.

Operand Stack

As shown in the previous addition example, in JAVA there may be intermediate values that are part of a computation, but which are not necessarily stored in a local variable.

Opcode	Description
3 ICONST_0	put integer constant 0 onto operand stack
21 ILOAD n	load integer from local variable n, put onto operand stack
54 ISTORE n	remove integer from operand stack, store into local variable n
96 IADD	replace two integers by their sum on the operand stack
153 IFEQ	remove integer number from operand stack, jump to specific target if number is 0
172 IRETURN	take integer value from operand stack, return it to invoking method
180 GETFIELD f	remove object reference from operand stack, put contents of a field f of referenced object onto operand stack
181 PUTFIELD f	remove object reference and another item from operand stack, put item into field f of referenced object
182 INVOKEVIRTUAL	take arguments from operand stack, invoke method
187 NEW c	Create instance of class c, put its reference onto operand stack

Figure 0.1.: Important Opcodes

By combining several individual computations, a large number of intermediate values needs to be part of the current state. In JAVA BYTECODE, in addition to local variables, the state contains an *operand stack* for such intermediate values. For example the opcode computing integer addition (IADD) takes both inputs *from* the operand stack and provides its result value *on* the operand stack. So, in order to store the sum of two local variables into another local variable, their contents first need to be put onto the operand stack and, after the addition, the result needs to be transferred back into a local variable.

Opcodes

Although for JAVA BYTECODE 255 different opcodes are defined, many of those differ only in minor aspects. For example, both the opcodes ILOAD_0 and ILOAD with the (hardcoded) argument 0 load an integer number from the first (0th) local variable. Furthermore, there exist opcodes ILOAD_1 to ILOAD_3. So, in total there are five different opcodes that each just load an integer number from a specific local variable. For the other data types (long, float, double, and object/array references) another 20 loading opcodes are defined. In Fig. 0.1 a small number of example opcodes is shown that can be used to illustrate most of the aspects dealt with in this thesis. The first column shows the index of each opcode and a more readable name which is also commonly used in the literature.

1. Symbolic Execution Graphs for Non-Recursive Programs

Programs written in JAVA or JAVA BYTECODE may make use of several techniques that make programming real-world programs easier, but on the other hand pose significant problems for program analysis. For example, in JAVA the target of a method invocation may be determined only at runtime if the programmer decided to make use of *method overriding* (dynamic dispatch). As a consequence, in order to be able to analyze such programs, the analysis must also correctly determine the targets of such method invocations and, therefore, stick to the rather involved semantics of the corresponding opcodes. It may be a good idea to develop an analysis that directly works on JAVA or JAVA BYTECODE programs, where complications like the one mentioned above are dealt with accordingly. However, with this approach it is not possible to directly benefit from already existing results in the area of static analysis.

The analysis presented in this thesis is part of the project AProVE [GBE$^+$14] which is also able to analyze *Term Rewrite Systems* (TRSs, [BN99, TeR03]). In particular, AProVE is able to analyze the termination behavior of TRSs using many techniques [Thi07, AG00], some dating back to 1979 [Lan79]. Instead of adapting or recreating the results already obtained for the termination analysis of TRSs, we decided to build on these results and transform JBC programs to TRSs in an appropriate way. This also means that the intricacies possible in JAVA also somehow need to be considered when creating such a TRS.

In the past, *Termination Graphs* were used for termination analysis of logic programs (written in PROLOG, [GSS$^+$12]) and functional programs (written in HASKELL, [GRS$^+$11]) to deal with language-specifics and create TRSs corresponding to the input programs. In this thesis, we will make use of this idea and construct *Symbolic Execution Graphs* for JAVA BYTECODE programs. These graphs deal with the language-specifics and also contain a lot of information that may be used for further analyses, for example termination analysis or bug detection. Recently, the ideas developed for the analysis of JAVA BYTECODE programs were adapted and extended to also analyze imperative programs making use of pointer arithmetic (written in C, [SGB$^+$14]).

An important aspect of the work presented in this chapter is that the analysis is fully automated. One only needs to provide the input program, there is no need to annotate parts of the program, give any sort of hint during analysis, or even define how heap abstraction should be performed. Using the implementation (part of the AProVE project, available at http://aprove.informatik.rwth-aachen.de) the claim of automated graph construction and subsequent termination analysis, performed by transforming the graph into term rewrite systems, can be verified by the interested reader.

Intuition

The main goal of this analysis is to construct a Symbolic Execution Graph for a Java Bytecode program $\mathcal{P}$ so that every computation possible in $\mathcal{P}$ can also be reproduced by following corresponding edges in the graph. A graph with this property then can be used to prove termination of $\mathcal{P}$ by showing that there is no infinite computation path in the graph. Also, because the graph contains an over-approximation of possible computations, one could show that specific undesired situations (e.g., throwing a certain exception) never occur in the program. The information in the graph can also help identifying certain bugs in the program.

To actually obtain a precise, yet finite representation of a possibly infinite number of computations possible for a (non-terminating) program, we use *symbolic execution* [Kin76] with abstraction. We start working on a start state containing only symbolic values for the inputs and then symbolically evaluate the program step by step. Using case analysis and abstraction in this process we finally obtain a finite graph with the properties mentioned above.

Structure

In the remainder of this chapter the details of this process are explained. First, we give an overview of related work in Section 1.1. In Section 1.2 we first explain the information needed to describe abstract states of the Java Virtual Machine. Then, in Section 1.3 we explain the idea of how to construct Symbolic Execution Graphs. The concept of *refinement*, which helps to provide the information necessary for evaluation which is not present in a state, is explained in Section 1.4. To create more precise information for one of the refinements, and as a preparation for a technique needed in Chapter 3, in Section 1.5 we introduce the concept of *state intersection*. In Section 1.6 we explain how evaluation of abstract states can be accomplished. Here, we especially consider the effects of write accesses. As creating a *finite* Symbolic Execution Graph is the goal of this analysis, in Section 1.7 we show how this goal is achieved. Combining all concepts of this chapter, in Section 1.8 we formally introduce the resulting Symbolic Execution Graphs and show properties which are useful for analyses making use of Symbolic Execution Graphs. Finally, in Section 1.9 we conclude and give an outlook on how this technique could be extended.

Limitations

While this analysis is able to deal with almost all features of JAVA BYTECODE, certain aspects are left for future work.

Floating-point numbers For variables declared as float or double the analysis only considers literals and the abstract value $\perp$ denoting unknown information. Hence, if the program behavior depends on the concrete value of such variables, the resulting graph contains significantly less precise information when compared to our handling of integer numbers.

Integers Integer numbers are treated as unbounded integers instead of machine-numbers with a limited size, as usual in program analysis.

Native Methods The code executed when invoking a native method is not part of the analyzed JAVA BYTECODE program and usually is written in a low-level language specific to the system the JAVA VIRTUAL MACHINE is running on. Therefore, analysis of such programs is out of scope and this technique does not work if native methods are invoked. However, certain native methods pre-defined in the standard classes of JAVA (e.g., java.lang.Throwable.fillInStackTrace) are correctly handled so that programs making use of common JAVA features can be analyzed.

Multithreading We only consider sequential programs.

Recursion In this chapter recursive programs may lead to a non-terminating graph construction (if the call stack can reach an arbitrary height). In Chapter 3 we will discuss necessary changes to also handle recursive programs.

Class objects The virtual machine may provide object instances of java.lang.Class using the native method getClass(). It is guaranteed that for each class the returned object instance is the same, in other words with X x1 = new X(); X x2 = new X(); we have x1.getClass() == x2.getClass(). Extending the analysis to maintain the necessary information is non-trivial, as the types of objects are not always known precisely. In Section 1.6.4 we explain how this approach can be extended to also handle Class objects correctly.

Interned Strings The JVM offers a way to return a unique java.lang.String object for each represented character sequence. If the character sequences in String variables s1 and s2 with s1 != s2 are identical, we have s1.intern() == s2.intern() (where intern() is a native method). As in the previous case, in Section 1.6.4 we explain how this approach can be extended to also handle interned String objects correctly.

Java version This technique was developed based on the specification corresponding to JAVA 6. As on a virtual machine level no significant changes are necessary to also support Java 7, we also implemented those changes (apart from INVOKEDYNAMIC, see below). However, this technique might need to be adapted to also support features introduced in newer versions of the Java language.

INVOKEDYNAMIC The opcode INVOKEDYNAMIC was added as part of the Java 7 release. The opcode is intended to be used by other (dynamic) languages that run on a Java Virtual Machine and, starting with Java 8, it is used for lambda expressions. This analysis is not able to work with programs containing this opcode.

1.1. Related Work

The analysis of this chapter is performed by symbolic execution with abstraction. In [Kin76] the basic idea of symbolic execution is presented. Building on this technique, the authors of [APV09] extend symbolic execution by also performing abstraction to ensure termination of the analysis. The graphs of [SG95] are created using similar concepts, where *driving* corresponds to evaluation and refinement as presented in this thesis, and *folding* and *generalization* correspond to how state instances are used in the analysis of this chapter. The technique of abstract interpretation [CC77] is very similar to our analysis. However, in our approach we do not formally define the abstract domain, but instead find abstract representations while we perform the analysis.

The approach of constructing graphs with the main goal of termination analysis has already been applied to the analysis of declarative programs. In [GSSKT06, GSS+12, GRS+11, Sch08] the authors not only present the graph construction, which directly corresponds to the results presented in this chapter. In addition to that, the authors present how term rewrite systems can be created based on the constructed graphs, such that termination of these TRSs implies termination of the original declarative program. The Symbolic Execution Graphs of this thesis can likewise be transformed to TRSs, as detailed in [OBvEG10, Bro14]. The idea of first constructing graphs which then can be transformed to TRSs with the goal of termination analysis has also been applied to imperative programs written in the language C, as explained in [SGB+14].

The challenge of finding a compact representation of arbitrary heaps is also addressed in research related to separation logic [ORY01, BCC+07, YLB+08, CRB10]. While the presented results allow for a very precise description of abstract heaps, automatically finding such representations is still an open problem.

There are tools explicitly designed for automated termination analysis of Java Bytecode. The tools Julia [SMP10] and COSTA [AAC+08] abstract the heap based on *path length*, which is a very simple abstraction. This allows for a very fast analysis, however the analysis presented in this thesis is more precise and powerful for user-defined data structures.

1.2. States

To model the states of computation used throughout the analysis, we closely follow the structure of states used in the JAVA VIRTUAL MACHINE. Each state consists of a call stack, containing individual stack frames (each with local variables and the operand stack), and a heap. The values on the heap are defined based on the possible values that may occur in a concrete evaluation. In the case of numbers we may store the literal value, and in the case of object instances and arrays it is possible to store the type and the contents of fields or array elements. Different from most JVM implementations we also place numbers on the heap and, just as for object instances and arrays, use references for these values. By also defining that return addresses (used for the obsolete opcodes JSR, JSR_W, and RET that were used to compile try-finally blocks of JAVA [LYBB12, §4.10.2.5]) are references, it suffices to only allow references as the contents of local variables, the operand stack, and fields.

We extend this model so that in addition to concrete states it is also possible to represent abstract states. For integer numbers intervals may be used to describe the possible values, although this could easily be extended to more precise abstract domains. For object instances it is possible to not define the value of certain (or all) fields, abstract the type, or describe abstract connections on the heap.

A very important assumption is that parts of the heap which are not represented explicitly are tree shaped (thus, also acyclic) and do not share with any other part of the heap. Here, we only consider sharing between object instances and arrays. In other words, two objects may contain a reference to the same integer number (or null) without being sharing. Arbitrary heaps can be modelled using *heap predicates* which will be explained in Section 1.2.2. Furthermore, for technical reasons we need another component containing a *split result*. Usage of this component will be explained in Section 1.4.

Definition 1.1 (Abstract States) A state is defined using several components. The first two components, call stack and heap, contain most of the actual data and are used in most opcodes.

$$\begin{aligned}\textsc{States} := {} & \textsc{CallStack} \times \textsc{Heap} \times \textsc{Types} \times \textsc{HeapPredicates} \times \\ & \textsc{StaticFields} \times \textsc{Exception} \times \textsc{InitializedClasses} \times \textsc{SplitResults}\end{aligned}$$

The call stack is composed of several stack frames for the currently running methods, where each stack frame contains local variables and an operand stack. We define that the first element in the CALLSTACK component is the topmost (current) stack frame.

$$\textsc{CallStack} := (\textsc{ProgramPositions} \times \textsc{LocalVariables} \times \textsc{OperandStack})^*$$

The set $\textsc{ProgramPositions}$ just contains a unique element for each opcode of the program, for example by combining the position of each opcode with the identifier of its method. Local variables and operand stacks are modelled using partial functions giving access to the references stored at some local variable or at some position in the stack. For the operand stack we define that the first element in the $\textsc{OperandStack}$ component of a stack frame is the topmost entry on the operand stack.

$$\begin{aligned}\textsc{LocalVariables} &:= \mathbb{N} \nrightarrow \textsc{References}\\ \textsc{OperandStack} &:= \mathbb{N} \nrightarrow \textsc{References}\end{aligned}$$

The set $\textsc{References}$ is infinite and contains the null reference in addition to all references that are used throughout the program. In contrast to languages like C, these references do not need to correspond to memory addresses, furthermore in Java it is not possible to read or modify references (although, of course the referenced data may be read and modified).

In addition to references used to reference data on the heap, the set $\textsc{References}$ also contains all *return addresses* which are used in the program. These return addresses are only used by the opcodes JSR, JSR_W, ASTORE*, and RET and (as a consequence that other opcodes may not be used for return addresses) may only be stored in a local variable or on the operand stack, but are never returned and never stored inside a field or array.

On the heap we distinguish between object instances and arrays, and integer and floating point numbers. We define that $\mathsf{null} \notin \mathrm{dom}(\textsc{Heap})$.

$$\begin{aligned}\textsc{Heap} &:= \textsc{References} \nrightarrow (\textsc{Instances} \cup \textsc{Arrays} \cup \textsc{Integers} \cup \textsc{Floats})\\ \textsc{Instances} &:= \textsc{FieldID} \nrightarrow \textsc{References}\\ \textsc{Arrays} &:= \textsc{References} \times (\mathbb{N} \nrightarrow \textsc{References})\\ \textsc{Integers} &:= \{\{x \in \mathbb{Z} \mid a \le x \le b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \le b\}\\ \textsc{Floats} &:= \mathbb{Q} \cup \{\mathsf{NaN}, -0, +\infty, -\infty\} \cup \{\bot\}\end{aligned}$$

By defining that each field identifier from the set $\textsc{FieldID}$ uniquely identifies a specific field (so it does not only reference the name of the field, but also the containing class and the field descriptor), we define data stored in the fields of object instances using a simple partial function. Together with the information about the type of an object instance this defines all aspects of an object instance. For each array we store a reference to the integer number which is the length of the array. Only if this number is a constant c, we may also store the contents of *all* field indices 0 to $c - 1$. Integers are defined using intervals, and for floating point values we only use a very limited abstract domain that only allows literals in addition to $\perp$ representing arbitrary values.

$$\textsc{Types} := \textsc{References} \nrightarrow 2^{\mathbb{N}\times(\textsc{PrimTypes}\ \cup\ \textsc{ClassNames})}$$
$$\textsc{PrimTypes} := \{\textsc{Boolean}, \textsc{Char}, \textsc{Float}, \textsc{Double}, \textsc{Byte}, \textsc{Short}, \textsc{Integer}, \textsc{Long}\}$$

The abstract type of an object instance or array is defined as a set of possible types. The number is the array depth of the type (0 if it is not an array). In case of an array depth of at least 1, the $\textsc{PrimTypes}$ component can be used to describe primitive arrays like int[]. Using the $\textsc{ClassNames}$ component individual classes (or arrays of classes) can be described, where $\textsc{ClassNames}$ contains all classes (and interfaces) of the program.

We limit the $\textsc{Types}$ component so that only types may be contained that correspond to the information in $\textsc{Heap}$. If h is the heap of the state, t is the $\textsc{Types}$ component and for a reference r we have $h(r) = f \in \textsc{Instances}$ we demand that $t(r) \subseteq 2^{\{0\}\times\textsc{ClassNames}}$ and all fields in $\mathrm{dom}(f)$ are defined in all classes of $t(r)$. Furthermore, we only consider states where the $\textsc{Types}$ component is not empty for any reference r referencing an object instance or an array. For $r = \mathsf{null}$ we define $t(r) = \emptyset$.

$$\textsc{StaticFields} := \textsc{FieldID} \nrightarrow \textsc{References}$$
$$\textsc{Exception} := \textsc{References} \cup \{\perp\}$$
$$\textsc{InitializedClasses} := \textsc{ClassNames} \rightarrow \{\textsc{Yes}, \textsc{No}, \textsc{Running}\}$$

Finally, static fields are defined globally and may be accessed from any part of the program. The exception component is used to denote which exception, if any, is thrown in the state. We use $\perp$ to denote that no exception is thrown. For each class we store its initialization state.

In Fig. 1.2 we show an abstract state for an arbitrary program. In this figure the most important components of Definition 1.1 are shown in an intuitive and more readable format.

$\langle 0 \,\vert\, \mathsf{this}\colon r_1, \mathsf{max}\colon i_1 \,\vert\, \varepsilon \rangle$
r_1: List(next: null, value: i_2) $i_1 : [0, \infty)$ $i_2 : (-\infty, \infty)$

Figure 1.2.: Abstract State

In the upper part, above the line, the call stack is shown. In this case, the call stack contains a single stack frame. In this stack frame, the first component indicates that the method is at the opcode at position 0. Then, the second component defines the local variables this and max of that stack frame[1]. For this the value r_1 is stored, for max we have the value i_1. The last component of the stack frame is the operand stack, which in this case is empty (as indicated by ε).

The lower part of the state defines the heap of the state. Here, we have that r_1 (which is the value of the local variable this) references a List object with fields named next and value, referencing null and i_2, respectively. For i_1 the abstract value $[0, \infty)$ is given, indicating that we just know that the value is non-negative. Similarly, for i_2 we have no further information, indicated by $(-\infty, \infty)$ (which could also be written as $\mathbb{Z}$).

Note that in this example we did not show how we represent arrays or floats on the heap. Furthermore, to simplify the presentation, we combined the type information (Type in Definition 1.1) and the information about fields of an instance (Instances). As such, for object instances with a more abstract type or fields defined in superclasses this simplified notation is not suitable. Last but not least, in this short example we have not included static fields, exceptions, information about initialized classes, and the split result.

1.2.1. Notation

In the course of this thesis references will usually be named r or r_x for some index x. In the case of references pointing to data from Integers, we usually use references of the form i_x. Furthermore, we will usually name the components of a state s as follows:

$$s = (\langle fr_0, \ldots, fr_n \rangle, h, t, hp, sf, e, ic, sr) \text{ with } fr_i = (pp_i, lv_i, os_i)$$

If not stated otherwise, with h and t we reference the heap resp. type information of state s. For a state with any name x we define that h_x is the heap component of x, while t_x is the type component of x. For example, if there is a state s' then $h_{s'}$ is the heap component of this state s'. We also have $h_s = h$, $t_s = t$. Furthermore, we also define that for a state s', $\overline{s}$, s_α, ... we identify the heap as h', $\overline{h}$, h_α, ... (respectively). Similarly, we use t' etc. for the type information of state s' etc.

We extend this even further. In contexts where we deal with a limited set of states that have disjoint sets of references, for a reference r we use s_r to identify the state containing r. Similarly, we use $h_r = h_{s_r}$ and $t_r = t_{s_r}$. To be more precise, null and all return addresses may appear in all states. However, for these references where h_r is not defined, we do not make use of the abbreviation. Further details will be explained before this notation is actually used.

[1] We added local variable names for simplicity, as in Java Bytecode local variables do not have names.

1.2.2. Heap Predicates

Using the components defined in Definition 1.1 only a limited number of abstract states can be represented. The assumption that, for example, sharing and cyclicity need to be represented explicitly makes it impossible to represent abstract states with these features. This problem is addressed by using *heap predicates*. The heap predicates defined in Definition 1.3 may be used to *allow* further connections between the given references. All heap predicates are limited to references to object instances or arrays, so sharing between an object and an integer number cannot be represented (and, since this connection is not considered to be sharing, this also is not needed). For example $(r_1, r_2) \in \textsc{PossibleEquality}$ is used to represent the states where r_1 and r_2 may reference the same object on the heap. Without this predicate, the assumption states that r_1 and r_2 point to different objects.

Definition 1.3 (Heap Predicates)

$$\begin{aligned}\textsc{HeapPredicates} \;:=\; & \textsc{PossibleEquality} \times \textsc{Joins} \times \\ & \textsc{Cyclic} \times \textsc{MaybeExisting}\end{aligned}$$

$$\begin{aligned}\textsc{PossibleEquality} &:= 2^{\textsc{References} \times \textsc{References}} \\ \textsc{Joins} &:= 2^{\textsc{References} \times \textsc{References}} \\ \textsc{Cyclic} &:= \textsc{References} \nrightarrow 2^{\textsc{FieldID}} \\ \textsc{MaybeExisting} &:= 2^{\textsc{References}}\end{aligned}$$

Both the predicates $\textsc{PossibleEquality}$ and $\textsc{Joins}$ are symmetric, i.e., $(r_i, r_j) \in \textsc{PossibleEquality} \Leftrightarrow (r_j, r_i) \in \textsc{PossibleEquality}$ and $(r_i, r_j) \in \textsc{Joins} \Leftrightarrow (r_j, r_i) \in \textsc{Joins}$. The predicate $\textsc{PossibleEquality}$ is irreflexive, i.e., $(o, o) \notin \textsc{PossibleEquality}$ for all $o \in \textsc{References}$. For every reference r marked as maybe existing in state s we demand that $h(r) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) = \emptyset$.

We do not allow $(o, o) \in \textsc{PossibleEquality}$ as we know that for any $o \in \textsc{References}$ the case that o references two different objects is not possible. In other words, allowing $(o, o) \in \textsc{PossibleEquality}$ would only allow for additional, impossible states.

The restriction that we do not retain any field information for object references marked using $\textsc{MaybeExisting}$ is helpful in ensuring termination of the graph construction.

Notation

Elements from PossibleEquality are written as $r_1 =^? r_2$. For Joins we use the notation $r_1 \searrow\!\swarrow r_2$. If $\text{Cyclic}(r_1) = F$ we write $r_1 \circlearrowleft_F$. The notation $r?$ is used for $r \in$ MaybeExisting.

In most cases the state containing the predicates is clear from the context. If not, a subscript is used to denote the corresponding state (e.g., $r_1 =^?_{s'} r_2$ for state s').

Intuition

In Fig. 1.4 the intuition for these heap predicates is shown. As explicit connections in a state may describe arbitrary shapes (including sharing, cyclicity, ...), the heap predicates are only used to describe properties of the parts of the heap which are not represented explicitly using connections of fields. Thus, when compared to the situation where heap predicates also need to be added for explicit connections, less heap predicates need to be introduced. This makes the analysis more precise. For example, if an object referenced by r_1 has a field f with the explicit information $r_1.f = r_1$, then for this cycle no heap predicate $r_1 \circlearrowleft_{\{f\}}$ needs to be used. However, if the field f is not represented explicitly in the state (i.e., it is not explicitly given what the content of the field $r_1.f$ is), one would need to use $r_1 \circlearrowleft_{\{f\}}$ to allow the case $r_1.f = r_1$.

In the following examples we assume that the heap predicates are used in an abstract state s and are used to describe connections in a concrete state c represented by s.

Heap Predicate	**Intuition**
$r_1 =^? r_2$	The objects in c corresponding to r_1 and r_2 may be identical.
$r_1 \searrow\!\swarrow r_2$	When following fields of the object corresponding to r_1 in c which are not represented explicitly in s, it is possible to reach an object in c corresponding to r_2 or an successor of this object. We use $r \searrow\!\swarrow r$ to indicate that r may reference a non-tree shape.
$r_1 \circlearrowleft_F$	In c there may be a cycle starting in an object corresponding to r_1 that uses at least one connection not explicitly represented in s. For each such cycle only fields defined in F are used.
$r?$	In c the corresponding object may not exist, i.e., the corresponding reference in c may be the `null` reference.

Figure 1.4.: Intuition for heap predicates

State Positions

To describe this intuition, the terms *successor* and *connection* are used, but not defined. As these concepts play an important role in the upcoming definitions and proofs, we will define *state positions*. A state position π for a reference r in state s describes a path in

s (starting in, e.g., some local variable) to r. We also write $s|_\pi = r$ if r is the reference at position π in state s. State positions can be used to describe connections between references.

Definition 1.5 (State Positions) Let $s = (\langle fr_0, \ldots, fr_n\rangle, h, t, hp, sf, e, ic, sr)$ be a state where each stack frame fr_i has the form (pp_i, lv_i, os_i). Then $\textsc{SPos}(s)$ is the smallest set containing all of the following sequences π:

- $\pi = \mathrm{LV}_{i,j}$ where $0 \le i \le n, lv_i = r_{i,0}, \ldots, r_{i,m_i}, 0 \le j \le m_i$. Then $s|_\pi$ is $r_{i,j}$.
- $\pi = \mathrm{OS}_{i,j}$ where $0 \le i \le n, os_i = r_{i,0}, \ldots, r_{i,m_i}, 0 \le j \le m_i$. Then $s|_\pi$ is $r_{i,j}$.
- $\pi = \mathrm{SF}_v$ where $sf(v) = r$. Then $s|_\pi$ is r.
- $\pi = \mathrm{EXC}$ where $e = r \neq \bot$. Then $s|_\pi$ is r.
- $\pi = \pi'\, v$ for some $v \in \textsc{FieldIDs}$ and some $\pi' \in \textsc{SPos}(s)$ where $h(s|_{\pi'}) = f \in \textsc{Instances}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.
- $\pi = \pi'\, i$ for some $i \in \mathbb{N}$ and some $\pi' \in \textsc{SPos}(s)$ where $h(s|_{\pi'}) = (i_l, f) \in \textsc{Arrays}$ and where $f(i)$ is defined. Then $s|_\pi$ is $f(i)$.
- $\pi = \pi'\, \mathsf{len}$ for $\pi' \in \textsc{SPos}(s)$ where $h(s|_\pi) = (i_l, f) \in \textsc{Arrays}$. Then $s|_\pi$ is i_l.

As the symbols $\sqsubseteq, \sqsubset, \ldots$ will be used for another prominent feature in this thesis, instead we use the symbols $\lhd, \unlhd, \unrhd, \rhd$ to denote (proper) prefixes and suffixes of positions:

- $\pi \lhd \pi'$ iff $\exists \tau \neq \varepsilon : \pi\tau = \pi'$
- $\pi \unlhd \pi'$ iff $\pi = \pi'$ or $\pi \lhd \pi'$
- $\pi \rhd \pi'$ iff $\pi' \lhd \pi$
- $\pi \unrhd \pi'$ iff $\pi' \unlhd \pi$

As an example, for the state shown in Fig. 1.2 we have the following state positions:

$$\{\mathrm{LV}_{0,0}, \mathrm{LV}_{0,1}, \mathrm{LV}_{0,0}\, \mathsf{next}, \mathrm{LV}_{0,0}\, \mathsf{value}\}$$

Using state positions it is now possible to formally define the terms *successor* and *connection* used before. A reference r_2 is a direct successor of r_1 in a state s if there are positions π, π' such that $s|_\pi = r_1, s|_{\pi'} = r_2$ and $\pi' = \pi\tau$ for some $\tau \in (\textsc{FieldIDs} \cup \mathbb{N} \cup \{\mathsf{len}\})$. A reference r_1 is connected to a reference r_2 using a path τ if there are positions

π, π' such that $s|_\pi = r_1, s|_{\pi'} = r_2$ and $\pi' = \pi\tau$ for some $\tau \in (\textsc{FieldIDs} \cup \mathbb{N} \cup \{\mathsf{len}\})^*$. For this we could also write $\pi \trianglelefteq \pi'$.

With this we can formally describe the meaning of heap predicates. This is done by defining which states are represented by an abstract state. We write $s' \sqsubseteq s$ to denote that s' is represented by s, which also means that all computations that are possible when starting in s' are also possible when starting in s. This *instance* definition is especially useful to define the relationship of abstract states to those states of a concrete evaluation (i.e., the states that are used in a computation of a real JVM) and will be used to prove corresponding properties of the constructed Symbolic Execution Graphs.

1.2.3. Concrete States

Before this instance definition is introduced, we first explain how concrete states are represented using states as defined in Definition 1.1. The definition of abstract states allows us to represent very precise information about states that may occur in some program evaluation. This information can even be so precise that no information is abstracted, i.e., the information contained in the state is identical to the in-memory information used by a real JVM for a real evaluation. In order to be able to define the relationship between the (abstract) states used in a Symbolic Execution Graph and a concrete evaluation, we define which subset of states is made up of *concrete states*.

Definition 1.6 (Concrete States) A state $s = (cs, h, t, hp, sf, e, ic, sr)$ is a concrete state if the following restrictions are met.

- $\forall r \in \textsc{References} \setminus \{\mathsf{null}\}$ where $t(r) = a$ is defined: $|a| = 1$. If $a \subseteq 2^{\{0\} \times \textsc{ClassNames}}$ then the only contained type is a non-abstract class.
- $\forall \pi \in \textsc{SPos}(s)$ with $h(s|_\pi) \in \textsc{Instances}$: $h(s|_\pi) = f$, $f(v)$ is defined for all field identifiers $v \in \textsc{FieldIDs}$ corresponding to the fields declared for the type $t(s|_\pi)$.
- $\forall \pi \in \textsc{SPos}(s)$ with $h(s|_\pi) \in \textsc{Arrays}$: $h(s|_\pi) = (i_l, f)$, $h(i_l)$ is some integer $l \geq 0$, $f(x)$ is defined for all $0 \leq x < l$.
- $\forall \pi \in \textsc{SPos}(s)$ with $h(s|_\pi) \in \textsc{Integers}$: $|h(s|_\pi)| = 1$.
- $\forall \pi \in \textsc{SPos}(s)$ with $h(s|_\pi) \in \textsc{Floats}$: $h(s|_\pi) \neq \bot$.
- No heap predicate exists in hp.
- $sr = \bot$.

1.2.4. State Instances

The following instance definition can be used to define which concrete states are represented by an abstract state. Furthermore, to obtain a finite Symbolic Execution Graph, we need to abstract the information stored for the individual states – this abstraction is done so that the more precise state is an instance of the more abstract state.

In the case that a long series of fields in a (more) concrete state s' is abstracted to a shorter series of fields in an abstract state s, we need to find out how much of it is realized in s. In other words, while $\pi \in \mathrm{SPos}(s')$ may hold, we may have that only a prefix of π is a valid position in s.

Definition 1.7 ($\overline{\pi}_s$) Let $s \in \textsc{States}$. Given a position π, $\overline{\pi}_s$ is the maximal prefix of π such that $\overline{\pi}_s \in \mathrm{SPos}(s)$ and for $\pi = \overline{\pi}_s\tau$ and $\varepsilon \lhd \tau' \unlhd \tau$ we have $s|_{\overline{\pi}_s\tau'} \notin \mathrm{SPos}(s)$. If the state is clear from the context, we just write $\overline{\pi}$. Note that we always have $\overline{\pi}_s \unlhd \pi$.

Furthermore, it may be the case that two positions point to the same reference according to their maximal existing prefixes in a state s, but where the remainder of the positions is identical.

Definition 1.8 (Suffixes of positions) Given a state s and two positions π, π' with $s|_{\overline{\pi}} = s|_{\overline{\pi'}}$ we say that π, π' have the same suffix w.r.t s iff for $\pi = \overline{\pi}_s\alpha$ and $\pi' = \overline{\pi'}_s\beta$ we have $\alpha = \beta$.

Similar to the definition of same suffixes, we also need to know if two positions have a common intermediate reference.

Definition 1.9 (Common Intermediate Reference) Let s be a state, α be a position and τ, τ' be two suffixes with $\{\alpha\tau, \alpha\tau'\} \subseteq \mathrm{SPos}(s)$. We define that τ, τ' have a common intermediate reference from α iff there are $\varepsilon \lhd \tilde{\tau} \lhd \tau$, $\varepsilon \lhd \tilde{\tau}' \lhd \tau'$ with $s|_{\alpha\tilde{\tau}} = s|_{\alpha\tilde{\tau}'}$.

Note that $s|_{\alpha\tau} = s|_{\alpha\tau'}$ is not considered to be a common intermediate reference.

With these auxiliary definitions we now define when exactly a state s' is an instance of a state s. As already mentioned, the intuition of this instance relation is that all computations possible in s' are also possible in s. Thus, it is important that any value represented in s' also is (at least implicitly) represented in s.

In Definition 1.10(a–c) we take care that s and s' have the same *shape*, i.e., the call stack has the same height, the opcodes are identical, both or none of the states throw an exception, and the initialization status if each class is identical.

In Definition 1.10(d–j) we show how the individual values may be abstracted. For return addresses no abstraction is possible, while the abstraction for floating point numbers is very limited. Integer values may be represented using an interval, thus we can, for example, concentrate on the sign of a number. The type of an object may be abstracted by allowing more than one type. As we may allow that a reference points to an existing object or null, we may abstract the null reference to a reference pointing to a possibly existing reference (for which no field values may be represented explicitly). Finally, abstracting instances and arrays is possible by representing less field/index information.

In Definition 1.10(k–r) we deal with the relationship of two references on the heap. Here, we basically add the aforementioned heap predicates if we decide to not explicitly represent certain heap shapes. In the upcoming proofs we will show that this choice indeed can be used to represent more abstract states.

Definition 1.10 ($\sqsubseteq$) Let $s' = (\langle fr'_0, \ldots, fr'_{n'}\rangle, h', t', hp', sf', e', ic', sr')$ and $s = (\langle fr_0, \ldots, fr_n\rangle, h, t, hp, sf, e, ic, sr)$, where $fr'_i = (pp'_i, lv'_i, os'_i)$ and $fr_i = (pp_i, lv_i, os_i)$. We call s' an *instance* of s (denoted $s' \sqsubseteq s$) iff

(a) $n = n'$ and $pp_i = pp'_i$ for all $0 \leq i \leq n$

(b) $e' = \bot$ iff $e = \bot$

(c) $ic' = ic$

For all $\pi \in \textsc{SPos}(s')$:

(d) if $s'|_\pi$ is a return address and $\pi \in \textsc{SPos}(s)$, then $s|_\pi = s'|_\pi$

(e) if $h'(s'|_\pi) \in \textsc{Floats}$ and $\pi \in \textsc{SPos}(s)$, then $h(s|_\pi) \in \{h'(s'|_\pi), \bot\}$

(f) if $h'(s'|_\pi) \in \textsc{Integers}$ and $\pi \in \textsc{SPos}(s)$, then $h'(s'|_\pi) \subseteq h(s|_\pi) \in \textsc{Integers}$

(g) if $t'(s'|_\pi)$ is defined and $\pi \in \textsc{SPos}(s)$, then $t'(s'|_\pi) \subseteq t(s|_\pi)$

(h) if $h'(s'|_\pi) = \mathsf{null}$ and $\pi \in \textsc{SPos}(s)$, then

- $h(s|_\pi) = \mathsf{null}$, or
- $s|_\pi?$ and $h(s|_\pi) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) = \emptyset$

(i) if $h'(s'|_\pi) = f' \in \textsc{Instances}$ and $\pi \in \textsc{SPos}(s)$, then $h(s|_\pi) = f \in \textsc{Instances}$ and $\mathrm{dom}(f') \supseteq \mathrm{dom}(f)$

(j) if $h'(s'|_\pi) = (i'_l, f') \in \textsc{Arrays}$ and $\pi \in \textsc{SPos}(s)$, then

- $h(s|_\pi) = (i_l, f) \in \textsc{Arrays}$ and $\mathrm{dom}(f') \supseteq \mathrm{dom}(f)$, or
- $h(s|_\pi) = f \in \textsc{Instances}$ and $\mathrm{dom}(f) = \emptyset$

$\langle 0 \mid \mathsf{this}\colon r_1, \mathsf{max}\colon i_1 \mid \varepsilon \rangle$
$r_1\colon \mathsf{List}(\mathsf{next}\colon r_2, \mathsf{value}\colon i_2)$
$r_2\colon \mathsf{List}(\mathsf{next}\colon \mathsf{null}, \mathsf{value}\colon i_3)$
$i_1 : [10, \infty)$
$i_2 : \{7\}$
$i_3 : (-\infty, 0]$

(a) State s_1

$\langle 0 \mid \mathsf{this}\colon r_3, \mathsf{max}\colon i_4 \mid \varepsilon \rangle$
$r_3\colon \mathsf{List}(\mathsf{next}\colon r_4, \mathsf{value}\colon i_5)$
$r_4 : \mathsf{List}()$
$i_4 : [0, \infty)$
$i_5 : (-\infty, \infty)$
$r_4?$

(b) State s_2

Figure 1.11.: States to illustrate $\sqsubseteq$

For all $\pi, \pi' \in \text{SPos}(s')$:

(k) if $s'|_\pi \neq s'|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(s)$, then $s|_\pi \neq s|_{\pi'}$

(l) if $h'(s'|_\pi) \in \textsc{Instances} \cup \textsc{Arrays}$, $\pi \neq \pi'$, $s'|_\pi = s'|_{\pi'}$, and $\pi, \pi' \in \text{SPos}(s)$, then $s|_\pi = s|_{\pi'}$ or $s|_\pi =^? s|_{\pi'}$

(m) if $h'(s'|_\pi) \in \textsc{Instances} \cup \textsc{Arrays}$, $\pi \neq \pi'$, $s'|_\pi = s'|_{\pi'}$ or $s'|_\pi =^? s'|_{\pi'}$, $\{\pi, \pi'\} \not\subseteq \text{SPos}(s)$, and $s|_{\overline{\pi}} \neq s|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. s, then $s|_{\overline{\pi}} \searrow\!\!\swarrow s|_{\overline{\pi'}}$

(n) if there are τ, τ', $\tau \neq \varepsilon$, α with $\pi = \alpha\tau$, $\pi' = \alpha\tau'$, τ, τ' have no common intermediate reference from α in s' (cf. Definition 1.9), $s'|_\pi = s'|_{\pi'}$, and $h'(s'|_\pi) \in \textsc{Instances} \cup \textsc{Arrays}$, then

- $\pi, \pi' \in \text{SPos}(s)$ and $s|_\pi = s|_{\pi'}$, or
- $\tau' \neq \varepsilon$ and $s|_{\overline{\alpha}} \searrow\!\!\swarrow s|_{\overline{\alpha}}$, or
- $\tau' = \varepsilon$ and $s|_{\overline{\alpha}} \searrow\!\!\swarrow s|_{\overline{\alpha}}$ and $s|_{\overline{\alpha}} \circlearrowleft_F$ with $F \subseteq \tau$ (where τ is interpreted as a set of field identifiers)

(o) if $s'|_\pi \circlearrowleft_{F'}$, then $s|_{\overline{\pi}} \circlearrowleft_F$ and $F \subseteq F'$

(p) if $s'|_\pi?$ and $\pi \in \text{SPos}(s)$, then $s|_\pi?$

(q) if $s'|_\pi =^? s'|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(s)$, then $s|_\pi =^? s|_{\pi'}$

(r) if $s'|_\pi \searrow\!\!\swarrow s'|_{\pi'}$, then $s|_{\overline{\pi}} \searrow\!\!\swarrow s|_{\overline{\pi'}}$

We illustrate Definition 1.10 by using two examples, which also explain some of the intuition behind the more complex parts of the definition.

Example 1.1 Assume we have the two states shown in Fig. 1.11. Here, in Fig. 1.11b we use $r_4?$ to indicate that r_4 is maybe existing (cf. Definition 1.3). Furthermore, we write r_4: $\mathsf{List}()$ to describe an object of type List where no information about its fields is represented. As such, the local variable this references a list of at least length one. Similarly, in Fig. 1.11a the list stored in the local variable this has exactly two elements, where the first element has a value of 7 and the second value is a non-positive number.

To see that we have $s_1 \sqsubseteq s_2$, we now consider the individual conditions of Definition 1.10. For that, we first see that Definition 1.10(a) holds as both states contain exactly one stack frame which is at the same program position. Assuming that the two states have no set exception reference and that the class initialization information also is identical, also Definition 1.10(b,c) hold.

For Definition 1.10(d–j) we need to consider all $\pi \in \mathrm{SPos}(s_1)$. We have $\mathrm{SPos}(s_1) = \{\mathrm{LV}_{0,0}, \mathrm{LV}_{0,1}, \mathrm{LV}_{0,0}\,\mathsf{next}, \mathrm{LV}_{0,0}\,\mathsf{value}, \mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next}, \mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{value}\}$. As we do not have return addresses, floats, or arrays in s_1, we do not need to consider Definition 1.10(d,e,j). First, consider $\pi = \mathrm{LV}_{0,1}$ with $s_1|_\pi = i_1$ and $h_1(i_1) = [10, \infty)$. According to Definition 1.10(f) we need to have $[10, \infty) \subseteq h_2(s_2|_\pi) \in \textsc{Integers}$. Indeed, as we have $h_2(s_2|_{\pi_2}) = [0, \infty)$, this holds. Similarly, the condition for $\mathrm{LV}_{0,0}\,\mathsf{value}$ is met as $h_1(i_2) = \{7\} \subseteq (-\infty, \infty) = h_2(i_5)$. For $\pi = \mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{value}$ we have $s_1|_\pi = i_3$, however we have $\pi \notin \mathrm{SPos}(s_2)$. Thus, we do not need to consider this position for Definition 1.10(f). The idea behind this is that parts of the heap which are not explicitly represented in the state may contain arbitrary data for integer fields.

We skip a detailed discussion of Definition 1.10(g,h) and instead explain Definition 1.10(i) in more detail. According to Definition 1.10(i) we only need to consider the positions $\mathrm{LV}_{0,0}$ and $\mathrm{LV}_{0,0}\,\mathsf{next}$. For $\pi = \mathrm{LV}_{0,0}$ we see that $h_1(s_1|_{\pi_1}) \in \textsc{Instances}$ and $h_2(s_2|_\pi) \in \textsc{Instances}$. Furthermore, we see that for both r_1 and r_3 the fields next and value are defined, thus we have $\mathrm{dom}(h_1(s_1|_\pi)) \supseteq \mathrm{dom}(h_2(s_2|_\pi))$. In the case of $\pi = \mathrm{LV}_{0,0}\,\mathsf{next}$ we see that in s_2 no field is defined, i.e., $\mathrm{dom}(h_2(s_2|_\pi)) = \emptyset$. Again, the intuition here is that parts of the state which are not explicitly represented may contain arbitrary data. Thus, the conditions of Definition 1.10(i) are also met for this position.

Finally, we consider Definition 1.10(k–r). We see that Definition 1.10(k) is trivially met. For Definition 1.10(l–r) we see that the preconditions are not met, as each object instance only has a single position, we do not have any non-tree shape, and we do not have any heap predicate in s_1.

To give a better insight into Definition 1.10(k–r), we now consider another example.

Example 1.2 We first consider Definition 1.10(k) and look at s_3 and s_5 as in Fig. 1.12a and Fig. 1.12c. We have $s_3|_{\mathrm{LV}_{0,0}} = r_1 \neq r_2 = s_3|_{\mathrm{LV}_{0,0}\,\mathsf{next}}$. Thus, according to Definition 1.10(k) we need to have $s_5|_{\mathrm{LV}_{0,0}} \neq s_5|_{\mathrm{LV}_{0,0}\,\mathsf{next}}$. The intuition is that having the same reference in two positions of a state means that in all represented (concrete) states the

$\langle 1 \mid \mathsf{this}: r_1 \mid r_1 \rangle$
r_1: $\mathsf{List}(\mathsf{next}: r_2, \mathsf{value}: i_1)$
r_2: $\mathsf{List}(\mathsf{next}: r_1, \mathsf{value}: i_1)$
$i_1 : (-\infty, \infty)$

(a) State s_3

$\langle 1 \mid \mathsf{this}: r_3 \mid r_4 \rangle$
$r_3 : \mathsf{List}(\mathsf{value}: i_2)$
$r_4 : \mathsf{List}(\mathsf{value}: i_2)$
$i_2 : (-\infty, \infty)$
$r_3 =^? r_4 \quad r_3 \curlyvee r_4$
$r_3 \curlyvee r_3 \quad r_4 \curlyvee r_4$
$r_3 \circlearrowleft_{\mathsf{next}} \quad r_4 \circlearrowleft_{\emptyset}$

(b) State s_4

$\langle 1 \mid \mathsf{this}: r_5 \mid r_5 \rangle$
$r_5 : \mathsf{List}(\mathsf{next}: r_5)$

(c) State s_5

Figure 1.12.: States to illustrate $\sqsubseteq$ w.r.t. heap predicates

corresponding object instances or arrays on the heap also are identical. However, this condition is not met. Thus, we have $s_3 \not\sqsubseteq s_5$.

Now consider s_3 and s_4 with s_4 as in Fig. 1.12b. We have $s_3|_{\mathrm{LV}_{0,0}} = s_3|_{\mathrm{OS}_{0,0}}$ and, with Definition 1.10(l) we need to have $s_4|_{\mathrm{LV}_{0,0}} = s_4|_{\mathrm{OS}_{0,0}}$ or $s_4|_{\mathrm{LV}_{0,0}} =^? s_4|_{\mathrm{OS}_{0,0}}$. Here the intuition is that the $=^?$ heap predicate may be used to describe aliasing on the heap which is not represented explicitly. As we have $r_3 =^? r_4$, this condition is met.

In Definition 1.10(m) we check if explicit sharing in s_3 for positions not represented in s_4 is allowed using joins heap predicates. In this case, we, for example, see that we have $s_3|_{\mathrm{LV}_{0,0}} = s_3|_{\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next}}$ where $\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next} \notin \mathrm{SPos}(s_4)$. Thus, we need to have $s_4|_{\overline{\mathrm{LV}_{0,0}}} \curlyvee s_4|_{\overline{\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next}}}$. As we have $\overline{\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next}}_{s_4} = \mathrm{LV}_{0,0}$ and we have $r_3 \curlyvee r_3$, this condition is met. Similarly, we need to have $r_4 \curlyvee r_4$ and $r_3 \curlyvee r_4$ in s_4, which all exist.

In Definition 1.10(n) we identify realized non-tree shapes in s_3 and demand that in s_4 such shapes either are represented explicitly, or allowed using the $\curlyvee$ and $\circlearrowleft$ heap predicates. As in the previous case, we have $s_3|_{\mathrm{LV}_{0,0}} = s_3|_{\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next}}$ where $\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next} \notin \mathrm{SPos}(s_4)$. Thus, we need to have $s_4|_{\overline{\mathrm{LV}_{0,0}\,\mathsf{next}\,\mathsf{next}}} \circlearrowleft_F$ for an appropriate value of F describing an underapproximation of the fields traversed along the cycle. Here, the value of $F = \emptyset$ would be sufficient, but the choice of $F = \{\mathsf{next}\}$ is also possible and more precise. Indeed, in s_4 we have $r_3 \circlearrowleft_{\mathsf{next}}$ and, thus, the condition is met. Similarly, we also have $r_4 \circlearrowleft_{\emptyset}$.

In Definition 1.10(o–r) we only take care that for existing heap predicates in s_3 we have correspondign heap predicates in s_4. As there are no heap predicates in s_3 and seeing that Definition 1.10(a–j) also hold, we conclude that $s_3 \sqsubseteq s_4$ holds.

For future proofs we need a lemma that describes the relationship of state positions in two states s, s' with $s' \sqsubseteq s$.

Lemma 1.3 (Positions in instances) Let $s, s' \in \textsc{States}$ with $s' \sqsubseteq s$. Then $\mathrm{SPos}(s) \subseteq \mathrm{SPos}(s')$.

Proof. Let $\pi \in \text{SPos}(s)$. We prove $\pi \in \text{SPos}(s')$ by induction on π. If $\pi = \text{LV}_{i,j}$ or $\pi = \text{OS}_{i,j}$, then the claim follows from the fact that $pp_i = pp'_i$ (Definition 1.10(a)) and that in verified JAVA BYTECODE, states that correspond to the same program position have the same local variables and the same number of entries on the operand stack. If $\pi = \text{EXC}$ then with Definition 1.10(b) also $\pi \in \text{SPos}(s')$. If $\pi = \text{SF}_v$ where c is the class of v then with Definition 1.10(c) we know that $ic(c) = ic'(c)$, thus $\pi \in \text{SPos}(s')$.

If $\pi = \pi' v$ for some $v \in \text{FieldIDs}$, then $h(s|_{\pi'}) = f \in \text{Instances}$ where $f(v)$ is defined. As $\pi' \in \text{SPos}(s)$, by the induction hypothesis, we know that $\pi' \in \text{SPos}(s')$ as well. Since $s' \sqsubseteq s$, $h(s|_{\pi'}) = f \in \text{Instances}$ and $f(v)$ is defined, Definition 1.10 implies $h'(s'|_{\pi'}) = f' \in \text{Instances}$ with $\text{dom}(f') \supseteq \text{dom}(f)$. Thus, $f'(v)$ is defined and $\pi' v = \pi \in \text{SPos}(s')$.

If $\pi = \pi' i$ for some $i \in \mathbb{N}$, then $h(s|_{\pi'}) = (i_l, f) \in \text{Arrays}$ where $f(i)$ is defined. As $\pi' \in \text{SPos}(s)$, by the induction hypothesis, we know that $\pi' \in \text{SPos}(s')$ as well. Since $s' \sqsubseteq s$, $h(s|_{\pi'}) = (i_l, f) \in \text{Arrays}$, Definition 1.10 implies $h'(s'|_{\pi'}) = (i'_l, f') \in \text{Arrays}$ with $\text{dom}(f') \supseteq \text{dom}(f)$. Thus, $f'(i)$ is defined and $\pi' i = \pi \in \text{SPos}(s')$.

If $\pi = \pi' \mathsf{len}$, then $h(s|_{\pi'}) = (i_l, f) \in \text{Arrays}$. As $\pi' \in \text{SPos}(s)$, by the induction hypothesis, we know that $\pi' \in \text{SPos}(s')$ as well. Since $s' \sqsubseteq s$ and $h(s|_{\pi'}) = (i_l, f) \in \text{Arrays}$, Definition 1.10 implies $h'(s'|_{\pi'}) = (i'_l, f') \in \text{Arrays}$. Thus, $\pi' \mathsf{len} = \pi \in \text{SPos}(s')$. $\square$

An important property of the $\sqsubseteq$ relation is transitivity.

Theorem 1.4 The $\sqsubseteq$ relation is transitive, i.e., for $s_1 \sqsubseteq s_2$ and $s_2 \sqsubseteq s_3$ we have $s_1 \sqsubseteq s_3$.

Proof. We show the claim by proving the individual items of Definition 1.10. It is important to note that for any $\pi \in \text{SPos}(s_3)$ we also have $\pi \in \text{SPos}(s_1) \cap \text{SPos}(s_2)$ due to Lemma 1.3.

(a – c) Trivial.

Let $\pi \in \text{SPos}(s_1)$. We also need to consider the case that $\pi \in \text{SPos}(s_3)$.

(d) Trivial.

(e) If $h_2(s_2|_\pi) = h_1(s_1|_\pi)$, the claim directly follows from $s_2 \sqsubseteq s_3$. Otherwise, if $h_2(s_2|_\pi) = \bot$ we also have $h_3(s_3|_\pi) = \bot$, thus the claim follows.

(f – g) The claim follows as $\subseteq$ is transitive.

(h) If $s_2|_\pi = s_1|_\pi = \mathsf{null}$, the claim directly follows from $s_2 \sqsubseteq s_3$. Otherwise, if $s_2|_\pi?$ and $h_2(s_2|_\pi) = f_2 \in$ Instances with $\mathrm{dom}(f_2) = \emptyset$, with Definition 1.10(i,p) we also have $s_3|_\pi?$ and $h_3(s_3|_\pi) = f_3 \in$ Instances with $\mathrm{dom}(f_3) \subseteq \mathrm{dom}(f_2) = \emptyset$. Thus, the claim follows.

(i) The claim follows as $\supseteq$ is transitive.

(j) Assume $h_1(s_1|_\pi) = (i_{l,1}, f_1) \in$ Arrays and $\pi \in \mathrm{SPos}(s_3)$. If $h_2(s_2|_\pi) = (i_{l,2}, f_2) \in$ Arrays with $\mathrm{dom}(f_2) \subseteq \mathrm{dom}(f_1)$, we also have $h_3(s_3|_\pi) = (i_{l,3}, f_3) \in$ Arrays with $\mathrm{dom}(f_3) \subseteq \mathrm{dom}(f_2)$ or $h_3(s_3|_\pi) = f_3 \in$ Instances with $\mathrm{dom}(f_3) = \emptyset$. Thus, the claim follows. Otherwise, we have $h_2(s_2|_\pi) = f_2 \in$ Instances with $\mathrm{dom}(f_2) = \emptyset$. Thus, the claim follows with Definition 1.10(i).

Let $\pi, \pi' \in \mathrm{SPos}(s_1)$.

(k) Trivial.

(l) We have $\pi, \pi' \in \mathrm{SPos}(s_3)$. From $s_1 \sqsubseteq s_2$ we conclude that $s_2|_\pi = s_2|_{\pi'}$ or $s_2|_\pi =^? s_2|_{\pi'}$ holds. With Definition 1.10(i,j) we also have that $h_2(s_2|_\pi) \in$ Arrays $\cup$ Instances. Thus, with $s_2 \sqsubseteq s_3$ and Definition 1.10(l,q) the claim follows.

(m) With Definition 1.10(m) and $s_1 \sqsubseteq s_2$ we may have $s_2|_{\overline{\pi}} \searrow\!\!\swarrow s_2|_{\overline{\pi'}}$. With Definition 1.10(r) and $s_2 \sqsubseteq s_3$ we then also have $s_3|_{\overline{\pi}} \searrow\!\!\swarrow s_3|_{\overline{\pi'}}$. If we do not have $s_2|_{\overline{\pi}} \searrow\!\!\swarrow s_2|_{\overline{\pi'}}$ then we know $s_2|_{\overline{\pi}} = s_2|_{\overline{\pi'}}$ where π, π' have the same suffix w.r.t. s_2. If we also have $s_3|_{\overline{\pi}} \neq s_3|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. s_3, then with $s_2 \sqsubseteq s_3$ we also have $s_3|_{\overline{\pi}} \searrow\!\!\swarrow s_3|_{\overline{\pi'}}$. Thus, the claim follows.

(n) Let $\pi = \alpha\tau$ and $\pi' = \alpha\tau'$ with $\tau \neq \varepsilon$ and τ, τ' have no common intermediate reference from α in s_1, and $h_1(s_1|_\pi) \in$ Instances $\cup$ Arrays. If $\{\pi, \pi'\} \not\subseteq \mathrm{SPos}(s_2)$ or $s_2|_\pi \neq s_2|_{\pi'}$, with Definition 1.10(n) and $s_1 \sqsubseteq s_2$ we have $s_2|_{\overline{\alpha}} \searrow\!\!\swarrow s_2|_{\overline{\alpha}}$. If $\tau' = \varepsilon$ we also have $s_2|_{\overline{\alpha}} \circlearrowleft_{F_2}$ with $F_2 \subseteq \tau$. Thus, with Definition 1.10(r,o) and $s_2 \sqsubseteq s_3$ we also have $s_3|_{\overline{\alpha}} \searrow\!\!\swarrow s_3|_{\overline{\alpha}}$ and (if $\tau' = \varepsilon$) $s_3|_{\overline{\alpha}} \circlearrowleft_{F_3}$ with $F_3 \subseteq F_2$.

Otherwise, assume $\pi, \pi' \in \mathrm{SPos}(s_2)$ and $s_2|_\pi = s_2|_{\pi'}$. Then, with Definition 1.10(n) and $s_2 \sqsubseteq s_3$ we also have $\pi, \pi' \in \mathrm{SPos}(s_3)$ and $s_3|_\pi = s_3|_{\pi'}$, or $s_3|_{\overline{\alpha}} \searrow\!\!\swarrow s_3|_{\overline{\alpha}}$ and (if $\tau' = \varepsilon$) $s_3|_{\overline{\alpha}} \circlearrowleft_{F_3}$ with $F_3 \subseteq \tau$.

(o) The claim follows as $\subseteq$ is transitive.

(p – r) Trivial. $\square$

1.3. Idea of Graph Construction

Using the concepts introduced so far, we now explain how to create a Symbolic Execution Graph. A Symbolic Execution Graph is always created in the context of a known Java Bytecode program. This means that the set ClassNames is fixed. In other words, the case that further classes may exist is not considered in the analysis.

Furthermore, a graph is constructed for a single abstract start state. This start state can be used, using a suitable abstraction, to represent several possible concrete start states. For example, it is possible to create a Symbolic Execution Graph for a method m that has a single integer argument, where all possible values for that argument are represented in the abstract start state.

Based on this start state, the graph is constructed using a simple fixed-point algorithm outlined in Algorithm 1. In Section 1.8 a more detailed description is presented. The main principle here is to evaluate the opcodes contained in each state contained in the graph, if possible. When evaluating an opcode of a state, a new state is added as a new leaf to the graph.

If evaluation is not possible, a case analysis based on the information of the state is performed so that in all resulting cases evaluation is possible. If, for example, a state contains $r?$ for some reference r and the opcode ISNULL needs to be evaluated for that reference, this is not possible. However, if the case analysis results in two states where in one state the reference r is replaced by null and in the other state r points to an existing object instance, evaluation is possible in those two states. In this thesis this kind of case analysis is named *refinement*.

The graph construction as just described may create an infinite graph. In order to always guarantee creation of a finite Symbolic Execution Graph, loops in a program are detected. By introducing more abstract states, for every loop after finite time a suitably abstract state is found that can be used to represent all (possibly infinitely many) loop iterations.

Example 1.5 We illustrate Algorithm 1 using the following simple Java program, which decreases the argument until it is negative.

```
public void someMethod(int arg) {
  while (arg >= 0) {
    arg--;
  }
  return;
}
```

The corresponding Java Bytecode also is quite simple. Here, the loop is realized

Algorithm 1: Graph construction

Input: $s_0 \in \textsc{States}$
Output: Symbolic Execution Graph $\mathcal{G}$
1: initialize $\mathcal{G}$
2: $\textsc{AddState}(\mathcal{G}, s_0)$
3: **for all** $s \in \textsc{LeafStates}(\mathcal{G})$ **do**
4:    **if** s is a repetition of s' **then**
5:       **if** $s \sqsubseteq s'$ **then**
6:          connect s to s' using an instance edge
7:       **else**
8:          $\textsc{ForceAbstraction}(s, s')$
9:    **else**
10:       **if** s can be evaluated **then**
11:          $\textsc{Evaluate}(s)$
12:       **else**
13:          $\textsc{Refine}(s)$

using the conditional branch IFLT 4 which jumps to line 4 if the topmost entry of the operand stack, in this case the value of the local variable arg, is less than 0. For non-negative values evaluation continues in line 2, where the value of arg is first decremented by 1 and then evaluation again continues at the loop head.

```
0    ILOAD arg
1    IFLT 4
2    IINC arg -1
3    GOTO 0
4    RETURN
```

We now explain how the Symbolic Execution Graph shown in Fig. 1.13 can be constructed for this program according to Algorithm 1. At first, we only have the start state A, which in the algorithm is named s_0. For this example we created the state such that the value of arg is arbitrary, so that the resulting Symbolic Execution Graph represents all possible program runs for any possible value of arg.

As A does not have any successors, it is a leaf state and the loop in lines 3 – 13 of Algorithm 1 is evaluated for state A. First, we see that A is no repetition of any other state, as it is the only state so far. Although the information in state A is quite abstract, we can evaluate the first opcode ILOAD arg, which just loads the value of arg into the operand stack. This evaluation results in state B, thus we connect A to B using an evaluation edge.

State B is a leaf state. We first see that B is no repetition of any other state, as the only other state A is at another position in the program (it is at opcode ILOAD arg in line 0, while state B is at opcode IFLT 4 in line 1). In state B we need to evaluate

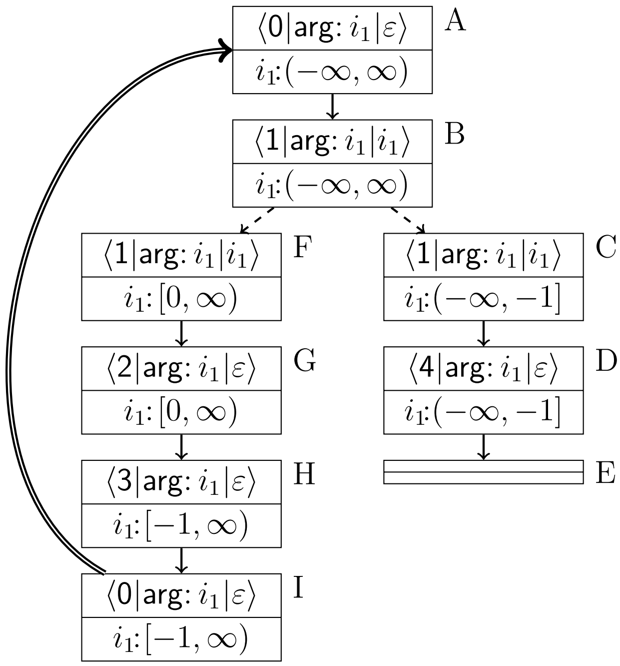


Figure 1.13.: A simple Symbolic Execution Graph

IFLT 4, which branches to line 4 if the value on top of the operand stack is negative, or continues in line 2 otherwise. As the information in state B does not give enough information to decide this, we need to refine B.

As the result of refinement we obtain two states C and F, which still are at the same opcode as state B. Both of these states are leaf states, and we first consider state C.

The information of state C still is abstract, but is precise enough to allow for an evaluation of IFLT 4. As we know that all values in the interval $(-\infty,-1]$ are smaller than 0, evaluation results in state D at opcode RETURN in line 4. Note that state C is not a repetition of any other state, especially not of states B or F, as we demand at least one evaluation edge on the path between two repeating states.

Evaluation of D finally leads to state E, which has an empty call stack. We do not consider states with an empty call stack as leaf states. Thus, the algorithm now only has to work on state F.

Similar to the evaluation of C to D, evaluation of state F results in state G. In G we need to evaluate IINC arg -1. We do not know the precise value of arg in state G, however we conclude that decrementing any value in the interval $[0,\infty)$ must result in a value in the interval $[-1,\infty)$. Thus, we obtain state H.

Evaluation of the GOTO opcode in state H is straight forward and results in state I. However, state I is at the same program position as state A. The path from state A to state I contains at least one evaluation edge, and thus I is a repetition of A.

According to Algorithm 1 we now check if $I \sqsubseteq A$ holds and, if this is not the case, enforce abstraction to ensure a finite graph construction. In this case $I \sqsubseteq A$ holds and, thus, we connect I to A using an instance edge.

Now no state of the graph is a leaf, resulting in termination of the graph construction.

While the idea of this algorithm is quite simple, three main aspects need to be regarded.

(i) Evaluation must be performed on abstract states.

(ii) It must be ensured that the states obtained using refinement cover all states represented by the unrefined state.

(iii) It must be ensured that the mentioned abstraction process indeed only is possible finitely often.

Without property (i) this technique could, at best, only be used as a Java Bytecode interpreter – which is uninteresting.

Without property (ii) a computation possible in the unrefined state may be impossible after refinement, making the resulting Symbolic Execution Graph useless for termination analysis (and other applications).

Without property (iii) we could not guarantee construction of the Symbolic Execution Graph in finite time, which severely limits the usefulness of this technique.

Furthermore, several technical challenges need to be solved. For example, Definition 1.10 must be regarded in the abstraction process. However, the definition does not give information about how to create a suitably abstract state as described above.

1.4. Refinement

The question whether a state can be evaluated is part of Algorithm 1. An intuition was already given before. In general we need to refine the information of the state so that the information necessary for (abstract) evaluation is represented. When dealing with opcodes working on integer numbers, for example IFEQ comparing an integer to 0, the information in the state may be too abstract in order to decide the outcome of the opcode to evaluate. In the example of IFEQ it can happen that a reference i_1 is used in the comparison, while in the heap the information $[0, \infty) \in \textsc{Integers}$ is stored for i_1. With this information it is possible that i_1 is 0, but it is also possible that this is not the case. In order to have a deterministic evaluation, we refine the information in the state and produce several states in this process, each of which then can be evaluated. In the example of IFEQ we could produce a state where the reference points to the integer constant 0, and another state where the integer is in the interval $[1, \infty)$.

However, there are also cases where it is not possible to construct a finite number of states so that each of these states can be evaluated. For example, consider the opcode

IF_ICMPEQ that compares two integer numbers for equality. If the intervals for these integers both contain infinitely many numbers, the idea presented above clearly cannot be used. Instead, we make use of the *split result* component, which is part of each abstract state. In this split result we encode the desired outcome of the opcode that should be evaluated. In the example of IF_ICMPEQ we would create two states, one with the split result encoding true (meaning that both integers are equal) and another state with the split result encoding false. In case no split result is needed, we use the split result $\perp$. Thus, even if the original state does not contain the information necessary for evaluation, the added split results enables us to evaluate.

To make sure that the final Symbolic Execution Graph represents all possible computations, we need to make sure that during refinement of s to $s_1, \ldots, s_n$ all concrete states c represented by the unrefined state (s) are also represented by at least one of the resulting states ($s_1, \ldots, s_n$). This leads us to the general definition what refinement is. The possible values of SPLITRESULT will be introduced in the course of this chapter.

Definition 1.14 (State Refinements) Let $s \in \textsc{States}$ and refine: $\textsc{States} \to 2^{\textsc{States}}$ be a refinement. This refinement is *valid* if and only if for all concrete states c with $c \sqsubseteq s$ a state $s' \in \text{refine}(s)$ exists with $c \sqsubseteq s'$.

In the course if this chapter, we will present several refinements. For all of these we will give proofs of validity according to Definition 1.14.

Note that for $\text{refine}(s) = \{s_1, \ldots, s_n\}$ there may be a concrete state c with $c \sqsubseteq s_i$ and $c \sqsubseteq s_j$ for $i \neq j$. Furthermore, we may have $c \sqsubseteq s_i$ where $c \not\sqsubseteq s$. However, the refinements presented in the remainder of this section are designed to be as precise as reasonably possible and, thus, avoid such outcomes.

To actually define different types of refinement, we need two auxiliary constructs that can be used to modify states. The first construct is used to replace references in a state by other references.

Definition 1.15 (Reference Replacement) $\sigma : \textsc{References} \to \textsc{References}$ is a *reference substitution* if the set $\{x \in \textsc{References} \mid \sigma(x) \neq x\}$ is finite. Let $s = (cs, h, t, hp, sf, e, ic, sr)$ be a state. We have

$$\sigma(s) = (\sigma(cs), h\sigma, t, hp, sf\sigma, \sigma(e), ic, sr)$$

where the application of σ to a call stack $cs = \langle f_0 \ldots, f_n \rangle$ is defined as follows:

$$\sigma(cs) = \langle f'_0, \ldots, f'_n \rangle \text{ with } f_i = (pp_i, lv_i, os_i) \text{ and } f'_i = (pp_i, lv_i\sigma, os_i\sigma) \text{ for } 0 \leq i \leq n$$

The individual values on the heap are replaced as follows:

$$h(r)\sigma = \begin{cases} f\sigma & \text{if } h(r) = f \in \textsc{Instances} \\ (\sigma(i_l), f\sigma) & \text{if } h(r) = (i_l, f) \in \textsc{Arrays} \\ h(r) & \text{otherwise} \end{cases}$$

For $s\sigma$ with $\sigma(r_i) = r'_i \quad (1 \le i \le m)$ we also write $s[r_1/r'_1, \ldots, r_m/r'_m]$.

The second construct is used to update values in or add values to a state.

Definition 1.16 (Heap Extension) Let s be a state. To change the value referenced on the heap, we use the notation $s + \{r \mapsto v\}$. We define that in $s + \{r \mapsto v\}$ all state components are identical to those of s, but the heap h is replaced by $h + \{r \mapsto v\}$:

$$(h + \{r \mapsto v\})(u) = \begin{cases} v & \text{if } r = u \\ h(u) & \text{otherwise} \end{cases}$$

Similarly, we also define $f + \{v \mapsto r\}$, and $t + \{r \mapsto T\}$ to modify field and type information, respectively:

$$(f + \{v \mapsto r\})(u) = \begin{cases} r & \text{if } v = u \\ f(u) & \text{otherwise} \end{cases}$$

$$(t + \{r \mapsto T\})(u) = \begin{cases} T & \text{if } r = u \\ t(u) & \text{otherwise} \end{cases}$$

1.4.1. Integer Refinement

We start by defining *integer refinement* which works by creating a partition of the integer value in question.

Definition 1.17 (Integer Refinement) Let $s \in \textsc{States}$ and let $r \in \textsc{References}$ with $h(r) = V \in \textsc{Integers}$.

Let $V_1, \ldots, V_n$ be a partition of V (i.e., $V_1 \cup \ldots \cup V_n = V$) with $\emptyset \neq V_i \subseteq \textsc{Integers}$. Moreover, $s_i = s + \{r \mapsto V_i\}$. Then $\text{refine}(s) = \{s_1, \ldots, s_n\}$ is an integer refinement of s.

In Fig. 1.13 you can see an example of integer refinement from node B to states C and F. Here, we have $(-\infty, \infty) = [0, \infty) \cup (-\infty, -1]$.

Theorem 1.6 Integer refinement is valid.

Proof. Let $\text{refine}(s) = \{s_1, \ldots, s_n\}$ be an integer refinement where $s_i = s + \{r \mapsto V_i\}$ and $h(r) = V = V_1 \cup \ldots \cup V_n \subseteq \mathbb{Z}$.

Let c be a concrete state with $c \sqsubseteq s$. Let $\Pi = \{\pi \in \text{SPos}(s) \mid s|_\pi = r\}$. By Definition 1.10(k) there is a $z \in V$ such that $h_c(c|_\pi) = \{z\}$ for all $\pi \in \Pi$. Let $z \in V_i$. Then $h_i(s_i|_\pi) = V_i$ for all $\pi \in \Pi$. To show $c \sqsubseteq s_i$ we only have to check Definition 1.10(f). Let $\tau \in \text{SPos}(s) \cap \text{SPos}(s_i)$ with $h_c(c|_\tau) = \{z'\} \in \text{Integers}$. If $\tau \notin \Pi$, then this position was not affected by the integer refinement and thus, $h_c(c|_\tau) \subseteq h_s(s|_\tau) = h_i(s_i|_\tau)$. If $\tau \in \Pi$, then we have $z' = z$ and thus $h_c(c|_\tau) \subseteq V_i = h_i(s_i|_\tau)$.

Corollary 1.7 For any integer refinement $\text{refine}(s) = \{s_1, \ldots, s_n\}$ and any concrete state c with $c \sqsubseteq s_i$ we have $c \sqsubseteq s$.

Proof. The claim holds because $V_i \subseteq V$. □

If using integer refinement the desired information cannot be provided, it may help to define the result using a *boolean split*. By extending of how an opcode is evaluated in the presence of a split result, the contained boolean value then provides the necessary information which cannot directly be obtained from the state. As an example, when evaluating IFLE for two unknown integer values, the split result true could indicate that the evaluation succeeds as described in the specification of the opcode. The details of such straightforward extensions are not given in this thesis.

Definition 1.18 (Boolean Split) Let $s \in \text{States}$. We demand that $\{\mathsf{true}, \mathsf{false}\} \subseteq \text{SplitResults}$. Let $s_{\mathsf{true}}, s_{\mathsf{false}}$ be identical to s where just the split result component is set to true resp. false. Then $\text{refine}(s) = \{s_{\mathsf{true}}, s_{\mathsf{false}}\}$ is a boolean split of s.

Theorem 1.8 Boolean split is valid.

Proof. Only the split result component of s is changed. This component is not considered in Definition 1.10, thus we have $s \sqsubseteq s_{\mathsf{true}}$ and $s \sqsubseteq s_{\mathsf{false}}$. With Theorem 1.4 the claim follows. □

Using integer refinement and boolean split it is possible to refine a state so that evaluation of most opcodes working on integers is possible. For the opcodes TABLESWITCH, LOOKUPSWITCH, and LCMP it may be necessary to use a split with more than two outcomes. A corresponding extension to Definition 1.18 is left as an exercise for the reader.

In general, it is always possible to provide the necessary information using splits. However, in order to have precise information in the resulting Symbolic Execution Graph, it is necessary to use refinements instead of splits if possible.

For example, for the code `if (x > 0) { a = b/x; }` first an integer refinement could be needed to evaluate `x > 0`. If instead of a refinement just a split is used, the resulting state used inside the body of the if statement computing the division would not contain the information that no division by zero is possible.

The details of how to find a suitable partition to perform an integer refinement are omitted in this thesis. Depending on the abstract domain used to represent Integers all kinds of optimizations could (and should) be performed in each refinement. The implementation in AProVE uses many optimizations so that as much information about integers is retained as feasible.

1.4.2. Existence Refinement

Similar to the test if an integer variable is 0, there also are opcodes that need to know if a reference is the null reference or not, i.e., if the referenced data actually exists. In our abstract representation of states a reference may be null (referenced data does not exist), the heap can map the reference to an element from Arrays and Instances (referenced data exists) or the heap predicate $r?$ is used for the reference r (existence of referenced data is unknown). If existence needs to be known for a reference r with $r?$ in the state, we perform *existence refinement.*

Definition 1.19 (Existence Refinement) Let s be a state and let r be a reference with $r?$ and $h(r) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) = \emptyset$. Then $\mathrm{refine}(s) = \{s_{\mathsf{null}}, s_{\mathsf{ex}}\}$ is an existence refinement where $s_{\mathsf{null}} = s[r/\mathsf{null}]$ and s_{ex} is a copy of s where the heap predicate $r?$ is removed.

Theorem 1.9 Existence refinement is valid.

Proof. Let r be the reference used in the refinement $\text{refine}(s) = \{s_{\mathsf{null}}, s_{\mathsf{ex}}\}$. Let c be a concrete state with $c \sqsubseteq s$. Because in an existence refinement the state is only altered at the positions $\Pi = \{\pi \mid s|_\pi = r\}$, we only need to consider these positions. From Definition 1.10(k) it follows that $c|_\pi = c|_{\pi'}$ for all $\pi, \pi' \in \Pi$. Consider any $\pi \in \Pi$. If $c|_\pi = \mathsf{null}$, then also $s_{\mathsf{null}}|_\pi = \mathsf{null}$. Thus, $c \sqsubseteq s_{\mathsf{null}}$ holds. Otherwise, $c|_\pi \neq \mathsf{null}$. As the only change from s to s_{ex} was the removal of $r?$, we only need to check Definition 1.10(h,p). As $c|_\pi \neq \mathsf{null}$ and no heap predicate $o?$ exists in c we conclude $c \sqsubseteq s_{\mathsf{ex}}$.

Corollary 1.10 Let $\text{refine}(s) = \{s_{\mathsf{null}}, s_{\mathsf{ex}}\}$ be an existence refinement. Then for any concrete state c with $c \sqsubseteq s_{\mathsf{null}}$ or $c \sqsubseteq s_{\mathsf{ex}}$ we have $c \sqsubseteq s$.

Proof. We first show $s_{\mathsf{null}} \sqsubseteq s$. s_{null} only differs from s in positions Π as defined in the proof of Theorem 1.9. For all $\pi \in \Pi$ we have $s_{\mathsf{null}}|_\pi = \mathsf{null}$ and $s|_\pi = r$ with $r?$ and $h(s|_\pi) = f \in \textsc{Instances}$ with $\text{dom}(f) = \emptyset$. Thus, $s_{\mathsf{null}} \sqsubseteq s$.

Finally, we show $s_{\mathsf{ex}} \sqsubseteq s$. s_{ex} is identical to s, but in s we additionally have $r?$. According to Definition 1.10 the heap predicate $r?$ may be added without consequences, so we have $s_{\mathsf{ex}} \sqsubseteq s$. □

1.4.3. Type Refinement

There are opcodes that depend on the type of a certain object instance or array. A simple example is the INSTANCEOF opcode, which checks if a given reference points to an object instance or array of a specific type. If the abstract type information is not precise enough to allow a evaluation of such opcodes, we perform *type refinement.*

Definition 1.20 (Type Refinement) Let $s \in \textsc{States}$ and let $r \in \textsc{References}$ with $\emptyset \neq t(r) = T \in \textsc{Types}$.

Let $T_1, \ldots, T_n$ be a partition of T (i.e., $T_1 \cup \ldots \cup T_n = T$) with $\emptyset \neq T_i \subseteq \textsc{Types}$. Moreover, s_i is a copy of s where the type information t is replaced by $t + \{r \mapsto T_i\}$. Then $\text{refine}(s) = \{s_1, \ldots, s_n\}$ is a type refinement of s.

Theorem 1.11 Type refinement is valid.

Proof. Let refine$(s) = \{s_1, \ldots, s_n\}$ be a *type refinement* of a reference r. Let $t(r) = T = T_1 \cup \ldots \cup T_n \in \textsc{Types}$.

Let c be a concrete state with $c \sqsubseteq s$. Let $\Pi = \{\pi \in \mathrm{SP}\textsc{os}(s) \mid s|_\pi = r\}$. By Definition 1.10(k) there is a $T_c \in T$ such that $t_c(c|_\pi) = \{T_c\}$ for all $\pi \in \Pi$. Let $T_c \in T_i$. Then $t_i(s_i|_\pi) = T_i$ for all $\pi \in \Pi$.

To show $c \sqsubseteq s_i$ we only have to check condition Definition 1.10(g). Let $\tau \in \mathrm{SP}\textsc{os}(s) \cap \mathrm{SP}\textsc{os}(s_i)$ with $t_c(c|_\tau) = T' \in \textsc{Types}$. If $\tau \notin \Pi$, then this position was not affected by the type refinement and thus, $t_c(c|_\tau) \subseteq t_s(s|_\tau) = t_i(s_i|_\tau)$. If $\tau \in \Pi$, then we have $T_c = T'$ and thus $t_c(c|_\tau) \subseteq T_i = t_i(s_i|_\tau)$.

Corollary 1.12 Let refine $s = \{s_1, \ldots, s_n\}$ be a type refinement. Then for any concrete state c with $c \sqsubseteq s_i$ we have $c \sqsubseteq s$.

Proof. The claim holds because $T_i \subseteq T$. □

1.4.4. Array Length Refinement

Another refinement is *array length refinement*. In our definition of abstract states we emphasized the length component which is part of every concrete array by also demanding that every abstract array has a reference to its length. This is motivated by the fact that the behavior of most algorithms working on arrays is determined by the length of the array instead of the actual contents. Opcodes working on arrays need to know if the referenced data is an existing array and they need to access the length of the array. For example, before an integer reference is read from an array, the corresponding IALOAD opcode first checks if the index used for the access is in the bounds of the array. While existence and type information can be provided using existence and type refinement, we now define array length refinement which can be used to provide the array length.

Definition 1.21 (Array Length Refinement) Let s be a state and r be a reference where no heap predicate $r?$ exists and $h(r) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) = \emptyset$ and for all $(l, c) \in t(r)$ we have $l > 0$ (i.e., we know that r references an array).

Let i'_l be a fresh reference. Then we define $\text{refine}(s) = \{s'\}$ where $s' = s + \{r \mapsto (i'_l, f') \in \textsc{Arrays}\} + \{i'_l \mapsto [0, \infty) \in \textsc{Integers}\}$ with $\text{dom}(f') = \emptyset$.

In other words, we replace information about an object without field information by an array with a reference to the length of the array, but still without information about the contents of the array.

Theorem 1.13 Array length refinement is valid.

Proof. Let $r \in \textsc{References}$ be the reference used in the refinement $\text{refine}(s) = \{s'\}$. Let c be a concrete state with $c \sqsubseteq s$. In an array length refinement the state is only altered at or below the positions $\Pi = \{\pi \mid s|_\pi = r\}$. From Definition 1.10(k) it follows that $c|_\pi = c|_{\pi'}$ for all $\pi, \pi' \in \Pi$. Consider any $\pi \in \Pi$. Because c is concrete and the type information of s denotes that r is an array, we know that $h_c(c|_\pi) = (i_{l,c}, f_c) \in \textsc{Arrays}$. In the refinement we only changed data at positions in Π or added a fresh reference in positions $\{\pi\mathsf{len} \mid \pi \in \Pi\}$. Therefore, we only need to check Definition 1.10(f,j). We have $h'(s'|_\pi) = (i'_l, f') \in \textsc{Arrays}$ with $\text{dom}(f') = \emptyset$. At positions $\{\pi\mathsf{len} \mid \pi \in \Pi\}$ we have $h_c(c|_{\pi\mathsf{len}}) = h_c(i_{l,c}) \in \textsc{Integers}$. We also have $h'(s'|_{\pi\mathsf{len}}) = h'(i'_l) \in \textsc{Integers}$ with $h_c(i_{l,c}) \subseteq h'(i'_l) = [0, \infty)$. Therefore, $c \sqsubseteq s'$.

Corollary 1.14 Let $\text{refine}\, s = \{s'\}$ be an array length refinement. Then we have $s' \sqsubseteq s$.

Proof. The additional information of the array length is of no consequence, as the additional positions for the array length reference are missing in s. According to Definition 1.10(j) we have $s' \sqsubseteq s$. □

1.4.5. Realization Refinement

Similar to array length refinement, we will introduce *realization refinement* now. This refinement is used if opcodes need to access a certain field in an object instance. If the abstract information of the referenced object instance does not provide any information about that field, we need to add this information to the state. While adding length information to an array is quite simple, in this case we also need to consider that fields may contain references to other object instances. Because of that we might need to introduce new heap predicates, for example to allow that a field references a cyclic data structure.

Note that it is also possible to define fields for which no non-abstract type exists (for example by defining a field with a type of an interface which is not implemented). In this case the refinement just returns a state where the field contains the null reference. This is correct since there also is no concrete state c with $c \sqsubseteq s$ (if s is the state for which we applied refinement) which contains a non-null value for that field.

Definition 1.22 (Realization refinement) Let v be a field identifier. Let s be a state and let r be a reference where no heap predicate $r?$ exists and $h(r) = f \in$ Instances. We demand that the field v is defined in each class class with $(0, \mathsf{class}) \in t(r)$. Furthermore, we demand that $f(v)$ is undefined.

Then refine$(s) = \{s'\}$ is a realization refinement with s' defined as follows. In all cases we introduce a new reference r' and define $f' = f + \{v \mapsto r'\}$.

If the type of v is a primitive $p \in \{\textsc{Boolean}, \textsc{Char}, \textsc{Byte}, \textsc{Short}, \textsc{Integer}, \textsc{Long}\}$ we define $s' = s + \{r \mapsto f'\} + \{r' \mapsto (-\infty, \infty)\}$.

If the type of v is a primitive $p \in \{\textsc{Float}, \textsc{Double}\}$ we define $s' = s + \{r \mapsto f'\} + \{r' \mapsto \bot\}$.

Otherwise, the type fieldtype of v is some class, interface, or array. Let $T_{\mathsf{fieldtype}} \subseteq$ Types be the abstract type that contains exactly all arrays and non-abstract classes which are subtypes of fieldtype. As noted above, if $T_{\mathsf{fieldtype}} = \emptyset$ we (re)define $r' = \mathsf{null}$, $t' = t + \{r' \mapsto \emptyset\}$ and disregard the following definitions of s'.

Otherwise, $T_{\mathsf{fieldtype}} \neq \emptyset$. Then we define a state s' with

- $h' = h + \{r \mapsto f' \in \textsc{Instances}\} + \{r' \mapsto f'' \in \textsc{Instances}\}$ where $\mathrm{dom}(f'') = \emptyset$
- $t' = t + \{r' \mapsto T_{\mathsf{fieldtype}}\}$
- $hp' = hp \cup \{r' \searrow r'', r' =^? r'' \mid r \searrow r''\} \cup \{r' \circlearrowleft_F \mid r \circlearrowleft_F\} \cup \{r' \searrow r' \mid r \searrow r\} \cup \{r'?\}$

Example 1.15 In Fig. 1.23 we illustrate Definition 1.22 using a short example. Here, we perform realization refinement on r_1 in state A where we realize the next field. In B we see the modified state where the next field of r_1 is set to a new reference r_3. For r_3 we provide the type information (indicated by r_3: List()).

As $r_1 \searrow r_2$ exists, it may be the case that the realized field content is identical to r_2, or that it itself may reach r_2. Thus, we need to add $r_3 \searrow r_2$ and $r_3 =^? r_2$. Furthermore, as we also have $r_1 \searrow\!\!\swarrow r_1$, we need to add $r_1 =^? r_3$, $r_1 \searrow\!\!\swarrow r_3$ and $r_3 \searrow\!\!\swarrow r_3$. Similarly, the $\circlearrowleft$ heap predicate is propagated, so that we have $r_3 \circlearrowleft_{\{\mathsf{next}\}}$. Finally, we allow the field content to be null by adding $r_3?$ to B.

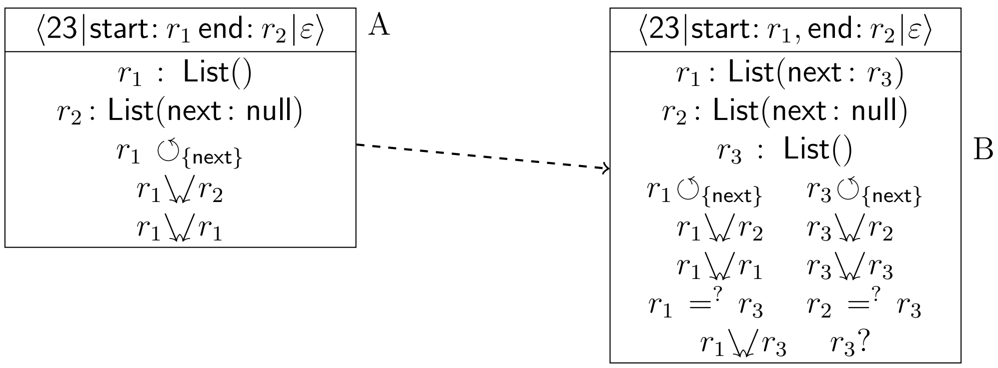


Figure 1.23.: Realization Refinement

Using Realization Refinement to provide the content of an array

Note that Definition 1.22 can easily be extended to the case where we want to refine a (literal) array index of an array with known size (i.e., both integer references point to integer intervals, each containing only a single literal). This case can be compared to realization refinement, where the array index is used instead of the field v. However, as for arrays we demand that the content function f either is empty (i.e., $\mathrm{dom}(f) = \emptyset$) or is defined for all array indices, this refinement would need to be adapted so that all indices are refined in a single step. The corresponding definition and proof are left as an exercise for the reader.

However, if the array index or the array length are not known to be literals, this is not possible. Instead, in Sections 1.6.2 and 1.6.3 we re-use the ideas presented in Definition 1.22 when defining how to evaluate the corresponding opcodes.

Theorem 1.16 Realization refinement is valid.

Proof. Let $\mathrm{refine}(s) = \{s'\}$ be a realization refinement on reference r. Let c be a concrete state with $c \sqsubseteq s$. Let $\Pi = \{\pi \mid s|_\pi = r\}$. Let $v \in \textsc{FieldIDs}$ be the refined field with type T_v. The realization refinement only changed values at and below positions in Π. It may have added heap predicates for references at other positions, but as heap predicates only *allow* more sharing effects, we do not have to consider these positions. By Definition 1.10(k), there is $f_c \in \textsc{Instances}$ such that $c|_\pi = f_c$ for all $\pi \in \Pi$. We prove $c \sqsubseteq s'$ by checking all conditions of Definition 1.10. Let $\mathcal{N} = \{\pi v \mid \pi \in \Pi\}$ be the positions that were newly created in the refinement. Let $\pi, \pi' \in \text{SPos}(c)$.

(a – d) Trivial.

(e) if $h_c(c|_\pi) \in \text{Floats}$ and $\pi \in \text{SPos}(s')$, then either
- $\pi \in \text{SPos}(s)$ and $h_c(c|_\pi) = h(s|_\pi) = h'(s'|_\pi)$ or $h(s|_\pi) = h'(s'|_\pi) = \bot$, or
- $\pi \in \mathcal{N}$ and thus $h'(s'_\pi) = \bot$

(f) if $h_c(c|_\pi) \in \text{Integers}$ and $\pi \in \text{SPos}(s')$, then either
- $\pi \in \text{SPos}(s)$ and $h_c(c|_\pi) \subseteq h(s|_\pi) = h'(s'|_\pi)$, or
- $\pi \in \mathcal{N}$ and thus $h'(s'_\pi) = (-\infty, \infty)$

(g) if $\pi \in \text{SPos}(s')$, then either
- $\pi \in \text{SPos}(s)$ and $t(s|_\pi) = t'(s'|_\pi)$, or
- $\pi \in \mathcal{N}$ and $t'(s'|_\pi)$ contains all non-abstract classes that may be stored in a field of type T_v. Hence, $t_c(c|_\pi) \subseteq t'(s'|_\pi)$.

(h) if $c|_\pi = \mathsf{null}$ and $\pi \in \text{SPos}(s')$, then either
- $\pi \in \text{SPos}(s) \setminus \Pi$ and thus the claim follows since $s|_\pi = s'|_\pi$, or
- $\pi \in \mathcal{N}$ and thus $s'|_\pi = \mathsf{null}$, or $h'(s'|_\pi)?$ and $h'(s'|_\pi) = f \in \text{Instances}$ with $\text{dom}(f) = \emptyset$

(i) if $h_c(c|_\pi) \in \text{Instances}$ and $\pi \in \text{SPos}(s')$, then either
- $\pi \in \text{SPos}(s) \setminus \Pi$ and $h(s|_\pi) = h'(s'|_\pi)$, or
- $\pi \in \Pi$ and $\text{dom}(h_c(c|_\pi)) \supseteq \text{dom}(h(s|_\pi)) = \text{dom}(h'(s'|_\pi)) \setminus \{v\}$. We also have $v \in \text{dom}(h_c(c|_\pi))$ because the field is defined in each type in $t(s|_\pi)$, or
- $\pi \in \mathcal{N}$ and thus $\text{dom}(h'(s'|_\pi)) = \emptyset$

(j) if $h_c(c|_\pi) \in \text{Arrays}$ and $\pi \in \text{SPos}(s')$, then either
- $\pi \in \text{SPos}(s) \setminus \Pi$ and $h(s|_\pi) = h'(s'|_\pi)$, or
- $\pi \in \mathcal{N}$ and thus $h'(s'|_\pi) = f' \in \text{Instances}$ with $\text{dom}(f') = \emptyset$

(k) In s', we only add new positions with null or a fresh reference which is different from all existing references. Hence, the claim follows from $c \sqsubseteq s$.

(l) if $c|_\pi = c|_{\pi'}$, $\pi \neq \pi'$, $h_c(c|_\pi) \in \text{Instances} \cup \text{Arrays}$, and $\pi, \pi' \in \text{SPos}(s')$, then either

- $\pi, \pi' \in \text{SPos}(s)$, and thus the claim follows since $s|_\pi = s'|_\pi$ and $s|_{\pi'} = s'|_{\pi'}$ and all heap predicates from s also exist in s', or
- exactly one of π, π' is in $\mathcal{N}$. W.l.o.g. let $\pi \in \mathcal{N}$, $\pi' \in \text{SPos}(s)$. Thus with Definition 1.10(m) we had $s|_{\overline{\pi}} \searrow\!\swarrow s|_{\pi'}$. Hence, in the refinement we added $s'|_\pi =^? s'|_{\pi'}$, or
- $\pi, \pi' \in \mathcal{N}$ – hence, $s'|_\pi = s'|_{\pi'}$.

(m) if $c|_\pi = c|_{\pi'}$, $\pi \neq \pi'$, $h_c(c|_\pi) \in \text{Instances} \cup \text{Arrays}$, and $\{\pi, \pi'\} \not\subseteq \text{SPos}(s')$, then either

- exactly one of $\overline{\pi}_{s'}, \overline{\pi'}_{s'}$ is in $\mathcal{N}$. W.l.o.g. let $\overline{\pi}_{s'} \in \mathcal{N}$. Hence, $\overline{\pi}_s \in \Pi$ and $\overline{\pi'}_s = \overline{\pi'}_{s'}$. With Definition 1.10(m) we have $s|_{\overline{\pi}} \searrow\!\swarrow s|_{\overline{\pi'}}$. Hence, in the refinement we added $s'|_{\overline{\pi}} \searrow\!\swarrow s'|_{\overline{\pi'}}$, or
- $\overline{\pi}_{s'}, \overline{\pi'}_{s'} \in \mathcal{N}$. Hence, $\overline{\pi}_s, \overline{\pi'}_s \in \Pi$. We have $s|_{\overline{\pi}} = s|_{\overline{\pi'}} = r$. If π, π' have the same suffix w.r.t. s, there is no need for a joins heap predicate in s'. Otherwise, we have $s|_{\overline{\pi}} \searrow\!\swarrow s|_{\overline{\pi'}}$ and, thus, also $s'|_{\overline{\pi}} \searrow\!\swarrow s'|_{\overline{\pi'}}$, or
- $\overline{\pi}_{s'} \notin \mathcal{N}, \overline{\pi'}_{s'} \notin \mathcal{N}$. Then also $\overline{\pi}_s = \overline{\pi}_{s'}$ and $\overline{\pi'}_s = \overline{\pi'}_{s'}$, and thus the claim follows since all heap predicates from s also exist in s'.

(n) Let $\pi = \alpha\tau$ and $\pi' = \alpha\tau'$ with $\tau \neq \varepsilon$, τ, τ' have no common intermediate reference from α in c, $h_c(c|_\pi) \in \text{Instances} \cup \text{Arrays}$, and $c|_\pi = c|_{\pi'}$. Then either

- $\pi, \pi' \in \text{SPos}(s)$, and thus the claim follows since $s|_\pi = s'|_\pi$ and $s|_{\pi'} = s'|_{\pi'}$ and all heap predicates from s also exist in s', or
- $\{\pi, \pi'\} \not\subseteq \text{SPos}(s)$ and $\overline{\alpha}_{s'} \in \text{SPos}(s)$. Then the claim follows since all heap predicates from s also exist in s'.
- $\{\pi, \pi'\} \not\subseteq \text{SPos}(s)$ and $\overline{\alpha}_{s'} \in \mathcal{N}$. Then $\overline{\alpha}_s \in \Pi$. Thus, we have $s|_{\overline{\alpha}} \searrow\!\swarrow s|_{\overline{\alpha}}$ and, if $\tau' = \varepsilon$, also $s|_{\overline{\alpha}} \circlearrowleft_F$ with $F \subseteq \tau$. Thus, the claim follows since these heap predicates are also propagated to $s'|_{\overline{\alpha}}$.

(o – r) Not applicable, as c is concrete. □

Corollary 1.17 Let refine $s = \{s'\}$ be a realization refinement. Then we have $s' \sqsubseteq s$.

Proof. Let $\Pi, \mathcal{N}$ be defined as in the proof of Theorem 1.16. To show the claim we do not need to regard positions $\pi \in (\text{SPos}(s) \cap \text{SPos}(s')) \setminus \Pi$, as the states are identical for these positions. Furthermore, for all positions $\pi \in \Pi$ the type information and all heap predicates were copied from s to s'. Because of that we only need to regard Definition 1.10(i, m, o, r).

(i) The entries $f \in$ INSTANCES are left unchanged or we add a field to the domain of f.

(m) W.l.o.g. we assume $\pi \in \mathcal{N}$, thus $\pi \notin \text{SPOS}(s)$ and $\overline{\pi}_s \in \Pi$.

If $\pi' \notin \mathcal{N}$ we also know $\pi' \in \text{SPOS}(s)$. We only need to consider the case that $s'|_\pi =^? s'|_{\pi'}$ (as we only add new references during the refinement). According to Definition 1.22 we added $s'|_\pi =^? s'|_{\pi'}$ only if $s|_{\overline{\pi}} \searrow\!\!\swarrow s|_{\pi'}$.

If $\pi' \in \mathcal{N}$ with $\pi \neq \pi'$ we know that $\overline{\pi}_s, \overline{\pi'}_s \in \Pi$ and $s|_{\overline{\pi}} = s|_{\overline{\pi'}}$. We also know that π, π' have the same suffix (namely v) w.r.t. s. Thus, there is no need for a joins heap predicate in s.

(o) W.l.o.g. we assume $\pi \in \mathcal{N}$. We only added $s'|_\pi \circlearrowleft_F$ if we had $s|_{\overline{\pi}} \circlearrowleft_F$ in s where $\overline{\pi}_s \in \Pi$.

(r) W.l.o.g. we assume $\pi \in \mathcal{N}$. We only added $s'|_\pi \searrow\!\!\swarrow s'|_{\pi'}$ if we had $s|_{\overline{\pi}} \searrow\!\!\swarrow s|_{\overline{\pi'}}$ in s where $\overline{\pi}_s \in \Pi$. $\square$

1.4.6. Equality Refinement

Finally, we need to introduce *equality refinement*. In abstract states it is possible for two references r, r' to point to the same object instance or array if $r =^? r'$ is set. For opcodes where evaluation is not possible if this heap predicate exists (e.g., IF_ACMPEQ), we create two states: one where r and r' point to different object instances/arrays, and one where both point to the same object instance/array. While creating a state for the first case is trivial, for the second state where both references point to the same object instance/array we would like to make use of the information we had for r and r' in the original state. In a simple example, we might know that r references an object instance of type X, while for r' we might know that the referenced object instance is acyclic. When combining this information, we could create a state containing a reference that points to an acyclic instance of type X. Furthermore, it is possible that no corresponding concrete state exists where r and r' are equal. This can happen, for example, if we have a cycle with $r = r'$ but where no such shape is allowed (i.e., the predicate $\circlearrowleft$ is missing).

In order to combine the information of such references r and r' and deal with *invalid states* as described above, we *intersect* the states obtained by replacing r by r' (i.e., $s[r/r']$) and replacing r' by r (i.e., $s[r'/r]$). Due to technical reasons, the references of the two intersected states need to be disjoint (apart from null and return addresses). Before we explain how to compute this intersection in Section 1.5, we define equality refinement.

Definition 1.24 (Equality Refinement) Let s be a state with $r =^? r'$, $h(r) \in$ $\textsc{Instances} \cup \textsc{Arrays}, h(r') \in \textsc{Instances} \cup \textsc{Arrays}$. We define $s_{\neq}$ as identical to s where just the heap predicate $r =^? r'$ is removed. The state $s_=$ is defined as $s_= = \text{intersect}(s[r/r'], \widetilde{s[r'/r]})$ where $\widetilde{s[r'/r]}$ is a renamed variant of $s[r'/r]$ such that the references in $s[r/r']$ and $\widetilde{s[r'/r]}$ are disjoint (apart from null and return addresses). If the intersection of the two states does not exist, we define $\text{refine}(s) = \{s_{\neq}\}$. Otherwise, $\text{refine}(s) = \{s_=, s_{\neq}\}$. Then $\text{refine}(s)$ is an equality refinement.

Theorem 1.18 Equality refinement is valid.

The proof is given in Section 1.5.5 on page 69.

In this section we defined several refinements. Each refinement can be used in situations where abstract evaluation is not possible. By performing case analyses we can create states that contain enough information so that in these states abstract evaluation can continue. For each refinement we have shown that it is valid: all concrete states represented by the unrefined state are also represented by at least one of the resulting states.

In Fig. 1.25 it is shown for which opcodes which refinements may be needed. Refinements marked with * indicate that a split may be needed. In the case of the integer refinement (e.g., needed for IF_IMPEQ) this was already motivated. For the AASTORE opcode which stores an object instance or array into an array, we may need to perform a type split. It is only possible to store a value of type Y into an array of type [X if Y is *assignment compatible* to X. For example, any object instance or array may be stored into an array of type [java.lang.Object, because every (non-primitive) type is an instance of java.lang.Object. However, there also are cases where even with a type refinement it is not possible to determine if the types are assignment compatible. Similar to the case of an integer split, the type of both the array and the type of the data to store into the array may be unknown, such that using non-split type refinement an infinite number of states would be needed. Instead, we perform a boolean split which indicates if the types are assignment compatible or not. As also an integer split may be needed to provide the necessary information to evaluate AASTORE, we combine integer splits and type splits into a single split.

Opcodes	Example	Refinement
46–53	IALOAD	existence, array length, integer*
79–82, 84–86	IASTORE	existence, array length, integer*
83	AASTORE	existence, array length, integer*, type*
108–115	IDIV	integer
148	LCMP	integer*
149–152	FCMPL	float*
153–158	IFEQ	integer
159–164	IF_IMPEQ	integer*
165–166	IF_ACMPEQ	equality
170	TABLESWITCH	integer*
171	LOOKUPSWITCH	integer*
180	GETFIELD	existence, realization
181	PUTFIELD	existence, realization, equality
182,185	INVOKEVIRTUAL	existence, type
183	INVOKESPECIAL	existence
188–189,197	NEWARRAY	integer
190	ARRAYLENGTH	existence, array length
191	ATHROW	existence
192–193	CHECKCAST	existence, type
198–199	IFNULL	existence

Figure 1.25.: Opcodes needing refinement

1.5. State Intersection

As motivated above, there are situations when we have the knowledge that two references point to the same object instance or array, while in the abstract state different pieces of information exist. Combining information from both of these states can make the analysis more precise. To this effect, we now introduce *state intersection*. While the definition and computation of state intersections is rather involved and there are other (less precise) methods that can be used for equality refinements (i.e., when refining $r_1 =^? r_2$ one could just replace r_1 by r_2 and, thus, "forget" about the information one has for r_1), we are interested in a precise analysis. Furthermore, to handle recursive programs (as explained in Chapter 3), state intersection as explained here is fundamentally important. Before we actually define state intersection, we introduce three main concepts.

(i) We first compute the references that must be considered to be identical. In the case of an equality refinement for a state s with $r_1 =^? r_2$ we enforce this by creating a state $s[r_1/r_2]$ and a state with $s[r_2/r_1]$. Thus, at least r_1 and r_2 must be considered to be identical. Depending on the state it is possible that also further references must point to the same information. As an example, consider a refinement of $r_1 =^? r_2$ where both r_1 and r_2 point to an object instance with a defined field v. Then, also $r_1.v$ and $r_2.v$ must be identical. To find references that must be considered to be identical, we compute a corresponding equivalence relation $\equiv$.

This equivalence relation is extended by finding out which references must actually be null because otherwise conflicts arise. If we have $r \equiv r'$, the represented values must be identical. However, if r and r' are in the same state and these values are object instances or arrays but the corresponding heap predicate $=^?$ is missing, the values can *only* be equal if both values are null. By exploiting missing but necessary $=^?$ heap predicates and performing other analyses, we extend $\equiv$ to $\equiv_{\mathsf{n}}$ were we merge the equivalence classes of null and (possibly) several references. This is helpful in obtaining a precise result.

(ii) In the next step we try to find conflicts which prevent intersection of the two states. This is the case if there is a reference $r \equiv_{\mathsf{n}}$ null where the heap predicate $r?$ does not exist.

(iii) Finally, we intersect the values contained in the state. For integer values this corresponds to the intersection of the intervals, in the case of object instances we consider the field information stored for all equivalent references.

1.5.1. Equivalence Relations $\equiv$, $\equiv_{\mathsf{n}}$

To identify references which must represent the same object instance, array, or primitive value, we define an equivalence relation $\equiv \subseteq$ REFERENCES $\times$ REFERENCES.

Definition 1.26 ($\equiv$) For two states s, s' let $\equiv \subseteq$ REFERENCES $\times$ REFERENCES be the smallest equivalence relation satisfying

(i) $\forall \pi \in \text{SPos}(s) \cap \text{SPos}(s') : s|_\pi = r \wedge s'|_\pi = r' \rightarrow r \equiv r'$

(ii) if $r \equiv r'$ and both $h_r(r) = f \in$ INSTANCES and $h_{r'}(r') = f' \in$ INSTANCES, then $f(v) \equiv f'(v)$ for all $v \in \text{dom}(f) \cap \text{dom}(f')$

(iii) if $r \equiv r'$ and both $h_r(r) = (i_l, f) \in$ ARRAYS and $h_{r'}(r') = (i'_l, f') \in$ ARRAYS, then $f(i) \equiv f'(i)$ for all $i \in \text{dom}(f) \cap \text{dom}(f')$ and $i_l \equiv i'_l$

The first item of Definition 1.26 describes the connection between two states. The other two items only are relevant if two references r, r' in the same state must be equivalent, both r and r' have some kind of successor, and all equivalent references in the other state do not have that successor. If the equivalent references in the other state had the successor, the first item already would suffice. However, as we allow $r =^? r'$ also in the case that there is detailed information for both r and r', the last two items help creating a better equivalence relation.

$\langle 23 \mid \mathsf{one}\colon r_1, \mathsf{two}\colon r_2 \mid \varepsilon \rangle$
$r_1 : \mathsf{List}(\mathsf{next} : r_3)$ $r_2 : \mathsf{List}(\mathsf{next} : r_1)$ $r_3 : \mathsf{List}()$

A

$\langle 23 \mid \mathsf{one}\colon r_4, \mathsf{two}\colon r_4 \mid \varepsilon \rangle$
$r_4 : \mathsf{List}(\mathsf{value} : i_1)$ $i_1 : [0, \infty)$

B

Figure 1.27.: Two states illustrating Definition 1.26

Example 1.19 Consider the two states from Fig. 1.27. If we compute $\equiv$ iteratively, at first we have the equivalence classes $\{r_1\}, \{r_2\}, \{r_3\}, \{r_4\}, \{i_1\}$. As we have $\mathrm{LV}_{0,0} \in \mathrm{SP\small{OS}}(A) \cap \mathrm{SP\small{OS}}(B)$ with $A|_{\mathrm{LV}_{0,0}} = r_1$ and $B|_{\mathrm{LV}_{0,0}} = r_4$, according to the first item we merge $\{r_1\}$ and $\{r_4\}$. Thus, we get the equivalence classes $\{r_1, r_4\}, \{r_2\}, \{r_3\}, \{i_1\}$. Similarly, with $\mathrm{LV}_{0,1}$ we merge the equivalence classes for r_2 and r_4, giving us $\{r_1, r_2, r_4\}, \{r_3\}, \{i_1\}$.

As these are the only two positions that exist in both states, we now consider the second item of the definition. Here, we see that we have $r_1 \equiv r_2$ where for both of the corresponding object instances we have a defined value for the next field. Thus, we merge the equivalence classes of r_3 and r_1. This finally gives us $\{r_1, r_2, r_3, r_4\}, \{i_1\}$ as the equivalence classes of $\equiv$.

Lemma 1.20 ($\equiv$ is sound) Let c be a concrete state with $c \sqsubseteq s$ and $c \sqsubseteq s'$. We assume that, apart from null and return addresses, s and s' have disjoint references. For all $r \equiv r'$ and positions π, π' with $s_r|_\pi = r$ and $s_{r'}|_{\pi'} = r'$ we have $c|_\pi = c|_{\pi'}$.

Proof. First we show that it suffices to have a single pair of positions $\pi \neq \pi'$ with $c|_\pi = c|_{\pi'}$, $s_r|_\pi = r$, and $s_{r'}|_{\pi'} = r'$ to show the claim. With such π, π', for any positions $\widetilde{\pi}, \widetilde{\pi}'$ with $s_r|_{\widetilde{\pi}} = r$ and $s_{r'}|_{\widetilde{\pi}'} = r'$, with Definition 1.10(k) we also have $c|_\pi = c|_{\widetilde{\pi}}$ and $c|_{\pi'} = c|_{\widetilde{\pi}'}$ as $c \sqsubseteq s_r$ and $c \sqsubseteq s_{r'}$. Thus, we also have $c|_{\widetilde{\pi}} = c|_\pi = c|_{\pi'} = c|_{\widetilde{\pi}'}$.

We show the claim by using an induction. In the base case, we may have $r = r'$ and $r \equiv r$. For any positions π, π' with $s_r|_\pi = s_{r'}|_{\pi'} = r = r'$ with Definition 1.10(d,h,k) and $c \sqsubseteq s, c \sqsubseteq s'$ we also have $c|_\pi = c|_{\pi'}$. Next, we consider $r \equiv r'$ because we have $\pi \in \mathrm{SP\small{OS}}(s) \cap \mathrm{SP\small{OS}}(s')$ and $s|_\pi = r$, $s'|_\pi = r'$. Let π_1, π_2 be positions with $s|_{\pi_1} = r$ and $s'|_{\pi_2} = r'$. As $s|_{\pi_1} = s|_\pi$, with Definition 1.10(k) and $c \sqsubseteq s$ we have $c|_{\pi_1} = c|_\pi$. Similarly, we have $c|_{\pi_2} = c|_\pi$. Thus, we also have $c|_{\pi_1} = c|_{\pi_2}$.

Now consider that we have $r \equiv r'$ because there are r_p, r'_p with $r_p \equiv r'_p$, $h_{r_p}(r_p) = f, h_{r'_p}(r'_p) = f'$, and $f(v) = r, f'(v) = r'$ for some $v \in \mathrm{F\small{IELD}IDs}$. By induction, we know

$c|_\pi = c|_{\pi'}$ for all π, π' with $s_{r_p}|_\pi = r_p$ and $s_{r'_p}|_{\pi'} = r'_p$. Thus, we also have $c_{r_p}|_{\pi v} = c_{r'_p}|_{\pi' v}$. The case involving Definition 1.26(iii) is analogous to the previous case.

Finally, we consider $r \equiv r'$ because we have $r \equiv r_m$ and $r_m \equiv r'$. With $r \equiv r_m$, by induction, we know $c|_\pi = c|_{\pi_m}$ for all π, π_m with $s_r|_\pi = r$ and $s_{r_m}|_{\pi_m} = r_m$. Similarly, we also have $c|_{\pi'} = c|_{\pi_m}$, where $s_{r'}|_{\pi'} = r'$. Combining this, we get $c|_\pi = c|_{\pi'}$. □

To further simplify the definition of state intersection, we now define which additional references are equivalent to null.

For that we need to check the reachability information contained in the states. Here, the intuition is that concrete connections in c either need to be represented in s and s', or must be implicitly allowed using heap predicates. To identify existing connections without having a concrete state c to look at, we make use of the information contained in the two states and combine this with the information of the equivalence relation. For this, we now define the $\stackrel{\tau}{\twoheadrightarrow}$ relation which holds if the described path must exist in all concrete states c with $c \sqsubseteq s$ and $c \sqsubseteq s'$.

Definition 1.28 ($\stackrel{\tau}{\twoheadrightarrow}$) Let $\equiv$ as in Definition 1.26, and let r be a reference with $h_r(r) \in$ Instances $\cup$ Arrays. Then we have $r' \stackrel{\tau}{\twoheadrightarrow} r$ iff one of the following conditions is met.

- We have $r' \equiv r$ and $\tau = \varepsilon$.
- We have $s_{r'}|_{\pi'} = r' \stackrel{\tau'}{\twoheadrightarrow} r_p = s_{r_p}|_{\pi_p}$, $s_{r_p}|_{\pi_p \tau''} \equiv r$ with $\tau'' \neq \varepsilon$, and $\tau = \tau'\tau''$.

$\langle 23 \mid \mathsf{one}: r_1, \mathsf{two}: r_2, \mathsf{three}: r_3 \mid \varepsilon \rangle$	A
$r_1 : \mathsf{List}(\mathsf{next} : r_2)$	
$r_2 : \mathsf{List}()$	
$r_3 : \mathsf{List}()$	

$\langle 23 \mid \mathsf{one}: r_4, \mathsf{two}: r_5, \mathsf{three}: r_6 \mid \varepsilon \rangle$	B
$r_4 : \mathsf{List}()$	
$r_5 : \mathsf{List}(\mathsf{next} : r_6)$	
$r_6 : \mathsf{List}()$	

Figure 1.29.: Two states illustrating Definition 1.28

Example 1.21 Consider $\equiv$ as computed for the two states of Fig. 1.29. First, we see that we have $r_1 \equiv r_4$, $r_2 \equiv r_5$, and $r_3 \equiv r_6$. Thus, we also have $r \stackrel{\varepsilon}{\twoheadrightarrow} r$ for all $r \in \{r_1, r_2, r_3, r_4, r_5, r_6\}$ and $r_1 \stackrel{\varepsilon}{\twoheadrightarrow} r_4$, $r_4 \stackrel{\varepsilon}{\twoheadrightarrow} r_1$, $r_2 \stackrel{\varepsilon}{\twoheadrightarrow} r_5$, $r_5 \stackrel{\varepsilon}{\twoheadrightarrow} r_2$, $r_3 \stackrel{\varepsilon}{\twoheadrightarrow} r_6$, and $r_6 \stackrel{\varepsilon}{\twoheadrightarrow} r_3$.

Furthermore, as we have $A|_{\mathrm{LV}_{0,0}} = r_1$ and $A|_{\mathrm{LV}_{0,0}\,\mathsf{next}} = r_2$ we get $r_1 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_2$. Similarly, we get $r_5 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_6$. By also considering equivalent references, this gives us $r_1 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_5$, $r_4 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_2$, $r_4 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_5$, $r_5 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_3$, $r_2 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_6$, and $r_2 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_3$.

More interestingly, as we have $r_1 \stackrel{\mathsf{next}}{\twoheadrightarrow} r_5$, $B|_{\mathrm{LV}_{0,1}} = r_5$, and $B|_{\mathrm{LV}_{0,1}\mathsf{next}} = r_6$, this also gives us $r_1 \stackrel{\mathsf{next}\,\mathsf{next}}{\twoheadrightarrow} r_6$, $r_4 \stackrel{\mathsf{next}\,\mathsf{next}}{\twoheadrightarrow} r_6$, $r_1 \stackrel{\mathsf{next}\,\mathsf{next}}{\twoheadrightarrow} r_3$, and $r_4 \stackrel{\mathsf{next}\,\mathsf{next}}{\twoheadrightarrow} r_3$.

Lemma 1.22 Let c, s, s' be states where $c \sqsubseteq s$, $c \sqsubseteq s'$, and c is concrete. Let s and s' have disjoint references apart from null and return addresses.

Then for all $s_{r'}|_{\pi'} = r' \overset{\tau}{\twoheadrightarrow} r = s_r|_\pi$ we have $c|_{\pi'\tau} = c|_\pi$.

Proof. We show the claim using an induction. Let $r' \overset{\tau}{\twoheadrightarrow} r$.

In the base case we have $r' \overset{\tau}{\twoheadrightarrow} r$ because we have $r' \equiv r$ and $\tau = \varepsilon$. Thus, the claim follows from Lemma 1.20.

We may also have $r' \xrightarrow{\tau'\tau''}\!\!\!\!\rightarrow r$ because we have $s_{r'}|_{\pi'} \overset{\tau'}{\twoheadrightarrow} r_p = s_{r_p}|_{\pi_p}$ with $s_{r_p}|_{\pi_p\tau''} \equiv r$. With Lemma 1.20 we have $c|_{\pi_p\tau''} = c|_\pi$. By induction we also have $c|_{\pi'\tau'} = c|_{\pi_p}$, thus also $c|_{\pi'\tau'\tau''} = c|_\pi$. $\square$

Using $\twoheadrightarrow$ we can now check s and s' if the corresponding connections are allowed. As the heap predicates only are used for connections on the heap which are not explicitly represented, we now define $\rightarrowtriangle$ to describe such connections based on $\twoheadrightarrow$.

Definition 1.30 ($\overset{\tau}{\rightarrowtriangle}$) Let $s_{r'}|_{\pi'} = r' \overset{\tau}{\twoheadrightarrow} r$ where $\tau \neq \varepsilon$ and $\pi' = \overline{\pi'\tau}_{s_{r'}}$. Then we have $r' \overset{\tau}{\rightarrowtriangle} r$.

As we have $r' \overset{\tau}{\rightarrowtriangle} r$ iff $r' \overset{\tau}{\rightarrowtriangle} \tilde{r}$ for all $\tilde{r} \equiv r$, we define that we have $r' \overset{\tau}{\rightarrowtriangle} [r]_\equiv$ if $r' \overset{\tau}{\rightarrowtriangle} r$.

We define that $r' \rightarrowtriangle r$ holds if there is any $\tau \neq \varepsilon$ with $r' \overset{\tau}{\rightarrowtriangle} r$.

Example 1.23 Let $\equiv$ be computed for the two states of Fig. 1.29.

According to Definition 1.30 we get $r_4 \xrightarrow{\mathsf{next}} r_5$, $r_4 \xrightarrow{\mathsf{next}} r_2$, $r_4 \xrightarrow{\mathsf{next\,next}} r_3$, $r_4 \xrightarrow{\mathsf{next\,next}} r_6$, $r_2 \xrightarrow{\mathsf{next}} r_3$, and $r_2 \xrightarrow{\mathsf{next}} r_6$.

Lemma 1.24 Let c, s, s' be states where $c \sqsubseteq s$, $c \sqsubseteq s'$, and c is concrete. Let s and s' have disjoint references apart from null and return addresses.

Then for all $s_{r'}|_{\pi'} = r' \overset{\tau}{\rightarrowtriangle} r = s_r|_\pi$ we have $c|_{\pi'\tau} = c|_\pi$, and $\pi' = \overline{\pi'\tau}_{s_{r'}}$.

Proof. For $r' \overset{\tau}{\rightarrowtriangle} r$ we have $r' \overset{\tau}{\twoheadrightarrow} r$. Thus, the claim follows with Lemma 1.22. $\square$

In the following lemma we now state that for any $\twoheadrightarrow$ connection we need to allow this connection using the joins heap predicate.

Lemma 1.25 Let c, s, s' be states where $c \sqsubseteq s$, $c \sqsubseteq s'$, and c is concrete. Let s and s' have disjoint references apart from null and return addresses.

Then for all $r' \twoheadrightarrow r = s_r|_\pi$ with $s_r = s_{r'}$ we have $r' \bigvee r$ or $c|_\pi = \mathsf{null}$.

Proof. Let $s_{r'}|_{\pi'} = r' \xrightarrow{\tau} r = s_r|_\pi$. With Lemma 1.24 we have $c|_{\pi'\tau} = c|_\pi$ and $\pi' = \overline{\pi'\tau_{s_{r'}}}$. Thus, if $s_r = s_{r'}$, with Definition 1.10(m) we have $s_{r'}|_{\pi'} = s_{r'}|_{\overline{\pi'\tau}} = r' \bigvee r = s_r|_\pi$ or $c|_\pi = \mathsf{null}$. $\square$

Now, we can define conditions under which references are known to be equivalent to null. The first check we perform is to consider the intersection of types. If two references point to the same object instance or array, the type of that referenced data needs to be contained in all abstract types. If this is not the case no such object can exist, i.e., the references must be equivalent to null.

Then, all references in the same equivalence class and the same state need to be pairwise connected using the $=^?$ heap predicate. Without this heap predicate, the referenced data must be a primitive (in INTEGERS or FLOATS) or the references must be equivalent to null.

We already explained the intuition for the third check. In essence, we make use of implicit reachability information.

Similar to the ideas presented so far, we also demand that for non-tree shapes which are known to exist in any concrete state c with $c \sqsubseteq s$ and $c \sqsubseteq s'$ the corresponding heap predicates exist.

Finally, we also demand that common abstract predecessors are marked as joining.

Definition 1.31 ($\equiv_{\mathsf{n}}$) Let $\equiv$ be an equivalence relation for states s, s' as defined in Definition 1.26. We assume that, apart from null and return addresses, s and s' have disjoint references.

We define $\equiv_{\mathsf{n}}$ based on $\equiv$ where we merge certain equivalence classes with the class for null.

(i) Let $r \equiv r'$ and $\{h_r(r), h_{r'}(r')\} \subseteq$ INSTANCES $\cup$ ARRAYS. If $t_r(r) \cap t_{r'}(r') = \emptyset$, then $\{r, r'\} \subseteq [\mathsf{null}]_{\equiv_{\mathsf{n}}}$.

(ii) Let $r \neq r'$ with $s_r = s_{r'}$, $r \equiv r' \not\equiv \mathsf{null}$, and $\{h_r(r), h_{r'}(r')\} \subseteq$ INSTANCES $\cup$ ARRAYS. If $r =^? r'$ does not exist, then $r, r' \in [\mathsf{null}]_{\equiv_{\mathsf{n}}}$.

(iii) Let $r' \rightarrowtriangle r$. Then we have $s_{r'} \neq s_r$, $r' \curlyvee r$, or $r \equiv_{\mathsf{n}}$ null.

(iv) Let $s_{r_1}|_{\pi_1} = r_1 \xrightarrow{\tau_1}\!\!\!\!\twoheadrightarrow r_2 = s_{r_2}|_{\pi_2}$ and $r_1 \xrightarrow{\tau_2}\!\!\!\!\twoheadrightarrow r_2' = s_{r_2'}|_{\pi_2'}$ with $r_2 \equiv r_2'$, $\varepsilon \neq \tau_1$, $\tau_1 \neq \tau_2$, and where τ_1, τ_2 have no corresponding intermediate reference from r_1 in s_{r_1}. We have $r_2 \equiv_{\mathsf{n}} r_2' \equiv_{\mathsf{n}}$ null if $\{\pi_1\tau_1, \pi_1\tau_2\} \not\subseteq \text{SPos}(s_{r_1})$ or $s_{r_1}|_{\pi_1\tau_2} \neq s_{r_1}|_{\pi_1\tau_2}$, and

- $r_1 \curlyvee r_1$ is missing, or
- $\tau_2 = \varepsilon$ and $r_1 \circlearrowleft_F$ with $F \subseteq \tau_1$ does not exist.

(v) Let $r_a \overset{\tau_a}{\rightarrowtriangle} r$ and $r_b \overset{\tau_b}{\rightarrowtriangle} r$ with $\tau_a \neq \tau_b$ or $r_a \neq r_b$. Then we have $s_{r_a} \neq s_{r_b}$, $r_a \curlyvee r_b$, or $r \equiv_{\mathsf{n}}$ null.

In the following example we illustrate when the third check helps us to identify a reference as being equivalent to null.

$\langle 23 \mid \mathsf{one}\colon r_1, \mathsf{two}\colon r_2 \mid \varepsilon\rangle$	A	$\langle 23 \mid \mathsf{one}\colon r_3, \mathsf{two}\colon r_4 \mid \varepsilon\rangle$	B	$\langle 23 \mid \mathsf{one}\colon r_5, \mathsf{two}\colon r_6 \mid \varepsilon\rangle$	C
r_1 : List(next : r_2) r_2 : List()		r_3 : List() r_4 : List()		r_5: List(value: i_1, next: r_6) r_6: List(value: i_1, next: null) i_1 : 0	

Figure 1.32.: Two states illustrating Definition 1.31(iii)

Example 1.26 For the states A and B of Fig. 1.32 we have $r_3 \xrightarrow{\mathsf{next}} r_4$. According to Definition 1.31(iii) we need to have $r_3 \curlyvee r_4$ or $r_4 \equiv_{\mathsf{n}}$ null.

Indeed, assume state C is a concrete state with $C \sqsubseteq A$ and $C \sqsubseteq B$. Then, as we have $C|_{\text{LV}_{0,0}\,\mathsf{next}} = C|_{\text{LV}_{0,1}}$ and $\text{LV}_{0,0}\,\mathsf{next} \notin \text{SPos}(B)$, according to Definition 1.10(m) we need to have $B|_{\overline{\text{LV}_{0,0}\,\mathsf{next}}} = B|_{\text{LV}_{0,0}} = r_3 \curlyvee r_4 = B|_{\overline{\text{LV}_{0,1}}} = B|_{\text{LV}_{0,1}}$.

As $r_3 \curlyvee r_4$ does not exist in B, we need to have $r_4 \equiv_{\mathsf{n}}$ null.

We now show that merging references into the equivalence class of null is correct, i.e., in all concrete states which are an instance of both input states the corresponding references indeed are null.

Lemma 1.27 ($\equiv_{\mathsf{n}}$ is sound) Let c be a concrete state with $c \sqsubseteq s$ and $c \sqsubseteq s'$. We assume that, apart from null and return addresses, s and s' have disjoint references. For all $r \equiv_{\mathsf{n}} r'$ and positions π, π' with $s_r|_\pi = r$ and $s_{r'}|_{\pi'} = r'$ we have $c|_\pi = c|_{\pi'}$.

Proof. As we only merged equivalence classes with the equivalence class $[\mathsf{null}]_\equiv$, it suffices to only consider r with $s_r|_{\pi} = r \equiv_{\mathsf{n}} r' = \mathsf{null}$ and show $c|_\pi = \mathsf{null}$. In all other cases the claim follows from Lemma 1.20. Furthermore, w.l.o.g. we assume that $r \not\equiv \mathsf{null}$. We only need to consider r with $h_r(r) \in \textsc{Instances} \cup \textsc{Arrays}$.

Let π with $s_r|_\pi = r$. We added r to $[\mathsf{null}]_{\equiv_\mathsf{n}}$ because of one of the following cases:

(i) Let $\tilde{r} \equiv r$ with $s_{\tilde{r}}|_{\tilde{\pi}} = \tilde{r}$. We have $t_r(r) \cap t_{\tilde{r}}(\tilde{r}) = \emptyset$. With Lemma 1.20 and $r \equiv \tilde{r}$ we know that $c|_\pi = c|_{\tilde{\pi}}$. Because of $c \sqsubseteq s$ and $c \sqsubseteq s'$ we know that $t_c(c|_\pi) = t_c(c|_{\tilde{\pi}}) \subseteq t(s_r|_\pi) \cap t_{\tilde{r}}(s_{\tilde{r}}|_{\tilde{\pi}}) = \emptyset$, thus $c|_\pi = \mathsf{null}$.

(ii) Let $\widetilde{\pi}, \tilde{r}$ as in the previous case. We have $r \equiv \tilde{r}$, $s_r = s_{\tilde{r}}$, and $r =^? \tilde{r}$ does not exist. With $r \equiv \tilde{r}$ and Lemma 1.20 we know $c|_\pi = c|_{\tilde{\pi}}$. With Definition 1.10(l) we conclude that $h_c(c|_\pi) \notin \textsc{Instances} \cup \textsc{Arrays}$, thus $c|_\pi = \mathsf{null}$.

(iii) The claim directly follows from Lemma 1.25.

(iv) Let $s_{r_1}|_{\pi_1} = r_1 \xrightarrow{\tau_1}\!\!\!\!\rightarrow r_2 = s_{r_2}|_{\pi_2}$ and $r_1 \xrightarrow{\tau_2}\!\!\!\!\rightarrow r_2' = s_{r_2'}|_{\pi_2'}$ with $r_2 \equiv r_2'$, $\varepsilon \neq \tau_1$, $\tau_1 \neq \tau_2$, and where τ_1, τ_2 have no corresponding intermediate reference from r_1 in s_{r_1}. With Lemma 1.22 we have $c|_{\pi_1\tau_1} = c|_{\pi_2}$ and $c|_{\pi_1\tau_2} = c|_{\pi_2'}$. With Lemma 1.20 we also have $c|_{\pi_2} = c|_{\pi_2'}$, thus also $c|_{\pi_1\tau_1} = c|_{\pi_1\tau_2}$.

If τ_1, τ_2 have a common intermediate reference from π_1 in c, let $\tau_1 = \tilde{\tau}_1\hat{\tau}_1$ with $\tilde{\tau}_1 \neq \varepsilon$ and $\tau_2 = \tilde{\tau}_2\hat{\tau}_2$ with $\tilde{\tau}_2 \neq \varepsilon$ such that $c|_{\pi_1\tilde{\tau}_1} = c|_{\pi_1\tilde{\tau}_2}$ and $\hat{\tau}_1, \hat{\tau}_2$ have no common intermediate reference from $\pi_1\tilde{\tau}_1$ in c. With Definition 1.10(l) we also have $s_{r_1}|_{\pi_1\tilde{\tau}_1} = s_{r_1}|_{\pi_1\tilde{\tau}_2}$, or $s_{r_1}|_{\pi_1\tilde{\tau}_1} =^? s_{r_1}|_{\pi_1\tilde{\tau}_2}$, or $\{\pi_1\tilde{\tau}_1, \pi_1\tilde{\tau}_2\} \not\subseteq \textsc{SPos}(s_{r_1})$. In the first case we have a contradiction.

In all other cases, also if no common intermediate reference in c exists as described above, the claim follows with Definition 1.10(n).

(v) Let $r_a \overset{\tau_a}{\rightarrowtriangle} r$ and $r_b \overset{\tau_b}{\rightarrowtriangle} r$ with $\tau_a \neq \tau_b$ or $r_a \neq r_b$. Let $s_{r_a}|_{\pi_a} = r_a, s_{r_b}|_{\pi_b} = r_b$, and $s_r|_\pi = r$. We also have $s_{r_a} = s_{r_b}$ and $r_a \bigvee\!\!\!\!/ r_b$ is missing.

With Lemma 1.24 we have $c|_{\pi_a\tau_a} = c|_\pi = c|_{\pi_b\tau_b}$. We also have $\pi_a = \overline{\pi_a\tau_a}_{s_a}$ and $\pi_b = \overline{\pi_b\tau_b}_{s_b}$. Thus, the claim follows with Definition 1.10(m).

Because of one of the cases, for $s_r|_\pi \equiv_\mathsf{n} \mathsf{null}$ we have $c|_\pi = \mathsf{null}$, thus the claim is shown.☐

Corollary 1.28 Let c be a concrete state. If $c \sqsubseteq s$ and $c \sqsubseteq s'$ then we have $c|_\pi = \mathsf{null}$ for all $\pi \in \{\pi \mid r \in [\mathsf{null}]_{\equiv_\mathsf{null}}, s_r|_\pi = r\}$.

1.5.2. Finding Conflicts

It may be the case that the information in the intersected states s, s' is conflicting in the sense that there cannot be a concrete state c with $c \sqsubseteq s$ and $c \sqsubseteq s'$. Continuing the graph construction with the intersection of s, s' is not necessary and might worsen the precision and usefulness of the whole analysis. Thus, we want to find such conflicts. The corresponding idea already was explained: each reference r that is equivalent to null either is null or it must have the $r?$ heap predicate. Furthermore, the referenced data may not be an array, and no field information may be given.

Definition 1.33 (Conflicts) Let s, s' be two states, let $\equiv_{\mathsf{n}}$ be a corresponding equivalence relation as defined in Definition 1.31. If there is a reference $r \equiv_{\mathsf{n}} \mathsf{null}$ where $r \neq \mathsf{null}$ and $r?$ is missing or $h_r(r) \in \textsc{Arrays}$ or $h_r(r) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) \neq \emptyset$, then the intersection of s and s' does not exist.

Theorem 1.29 If according to Definition 1.33 no intersection of s, s' exists, then there is no concrete state c with $c \sqsubseteq s$ and $c \sqsubseteq s'$.

Proof. According to Corollary 1.28 we have $c|_\pi = \mathsf{null}$ for all π with $s_r|_\pi = r$ and $r \in [\mathsf{null}]_{\equiv_{\mathsf{n}}}$. Thus, with $c \sqsubseteq s_r$ and Definition 1.10(h) we have $r = \mathsf{null}$ or $r?$ with $h(r) = f \in \textsc{Instances}$ and $\mathrm{dom}(f) = \emptyset$. This is not the case in s_r, thus $c \not\sqsubseteq s_r$. $\square$

Example 1.30 Again, consider the states of Fig. 1.32. We already have demonstrated that $r_4 \equiv_{\mathsf{n}} \mathsf{null}$ holds.

However, as $r_4?$ does not exist in B, we conclude that there is no state c with $c \sqsubseteq A$ and $c \sqsubseteq B$.

1.5.3. Intersecting Values

In order to define the state $\overline{s}$ that is the result of intersecting s and s', we need to intersect the values in the states. Furthermore, we need to use a single reference that is used in place of all references of the same equivalence class.

We first define an injective function ρ that gives a unique reference for each equivalence class. Based on ρ we define a reference replacement σ that replaces any reference r with the unique reference for $[r]_{\equiv_{\mathsf{n}}}$. However, we do not want to replace null or return addresses and we also want to replace all references in the same equivalence class as null by null.

Definition 1.34 ($\boldsymbol{\rho, \sigma}$) Let $\equiv_{\mathsf{n}}$ be an equivalence relation for states s, s' as defined in Definition 1.31. Let $\textsc{References}(s)$ (resp. $\textsc{References}(s')$) be the references of s (resp. s'). For each equivalence class $[r]_{\equiv_{\mathsf{n}}}$ where $r \not\equiv \mathsf{null}$ and r is no return address, we define that $\rho([r]_{\equiv_{\mathsf{n}}})$ is a fresh reference so that $\rho(r) \notin \textsc{References}(s) \cup \textsc{References}(s')$. For $[\mathsf{null}]_{\equiv_{\mathsf{n}}}$ we define $\rho([\mathsf{null}]_{\equiv_{\mathsf{n}}}) = \mathsf{null}$, and for each equivalence class that contains return addresses we define that ρ maps to one of these return addresses (we later ensure that only a single return address is contained in each such equivalence class). Let σ be the *identification substitution* which maps equivalent references to the same fresh reference (and non-equivalent references to different references): $\sigma(r) = \rho([r]_{\equiv_{\mathsf{n}}})$ if $r \in \textsc{References}(s) \cup \textsc{References}(s')$. If we have $r = \rho([r']_{\equiv_{\mathsf{n}}})$, for any such r, r' we define $\sigma(r) = r$.

To ease the definition of state intersection, we also define some auxiliary functions. The first of these is used to combine the field information we have for two object instances (or to combine the array information of two arrays).

Definition 1.35 ($\boldsymbol{f \cup f'}$) Let f, f' be two functions with the same signature. Then we define the function $f \cup f' := f_{f \cup f'}$ with the same signature:

$$f_{f \cup f'}(x) = \begin{cases} f(x) & \text{if } x \in \mathrm{dom}(f) \\ f'(x) & \text{if } x \notin \mathrm{dom}(f) \wedge x \in \mathrm{dom}(f') \end{cases}$$

To intersect states, we may need to intersect several values. We define binary intersections using $\Cap$ and use these binary intersections to intersect an arbitrary number of values using $\bigcap$. The symbol ※ is used as a result for values that cannot be intersected.

Definition 1.36 ($\Cap$) Let $\textsc{Values} := \textsc{Instances} \cup \textsc{Arrays} \cup \textsc{Integers} \cup \textsc{Floats} \cup \{※\}$. The intersection $\Cap\colon \textsc{Values} \times \textsc{Values} \to \textsc{Values}$ is defined as follows:

$$v \Cap v' = \begin{cases} v \cap v' & \{v, v'\} \subseteq \textsc{Integers} \wedge v \cap v' \neq \emptyset \\ v & \{v, v'\} \subseteq \textsc{Floats} \wedge (v = v' \vee v' = \bot) \\ v' & \{v, v'\} \subseteq \textsc{Floats} \wedge v \neq v' \wedge v = \bot \\ (\sigma(i_l), (f \cup f')\sigma) & \{v, v'\} \subseteq \textsc{Arrays} \wedge v = (i_l, f) \wedge v' = (i'_l, f') \\ (\sigma(i_l), f\sigma) & v = (i_l, f) \in \textsc{Arrays} \wedge v' = f' \in \textsc{Instances} \wedge \operatorname{dom}(f') = \emptyset \\ (\sigma(i'_l), f'\sigma) & v = f \in \textsc{Instances} \wedge \operatorname{dom}(f) = \emptyset \wedge v' = (i'_l, f') \in \textsc{Arrays} \\ (f \cup f')\sigma & \{v, v'\} \subseteq \textsc{Instances} \wedge v = f \wedge v' = f' \\ ※ & \text{otherwise} \end{cases}$$

With this definition we intersect two integer intervals (if possible) or retain a float literal. For arrays we combine the information we have about the contents of the arrays by considering the union of the corresponding functions. Note that it does not matter which reference is used for the length (i.e., i_l or i'_l), since we only intersect values of equivalent references and we have $i_l \equiv_{\mathsf{n}} i'_l$. If both an array and an instance are referenced by r and r', the intersection only exists if the referenced object instance does not define any field (e.g., it may be java.lang.Object). In this case, the intersected value just is the array. Finally, if we intersect two object instances, we combine the information we have about the fields by considering the union of the corresponding functions, similar to the case of arrays described above.

1.5.4. Intersecting States

Now we can finally define how to intersect two states s, s'. First, we assume that $\equiv_{\mathsf{n}}$ already is computed and that no conflict exists according to Definition 1.33. The basic idea in this intersection process is to intersect the values of each equivalence class.

We only add a heap predicate if the corresponding information is represented in both input states, for all combinations of equivalent references. For example, we only add $\sigma(r) =^? \sigma(r')$ to the intersected state, if all references in the equivalence classes of r and r' are also marked as possibly equal. To obtain more precise results, we also make sure that the information of abstract predecessors (w.r.t. $\multimap$) matches. Thus, there is no need to add $\sigma(r) =^? \sigma(r')$ if an abstract predecessor of r may not reach a reference in the same state which is equivalent to r'. Some of these constraints on when we need to add a heap predicate may look like superfluous optimizations. However, as state intersection is used in Chapter 3 where very precise results are needed for correctness, we decided to also mention the corresponding changes here. In this slightly simpler setting (compared to Chapter 3) it is easier to understand the reasoning behind those optimizations, which makes it easier to understand the necessary changes in Chapter 3.

As we often need to consider references which either are in a specific equivalence class, or are sure to reach it (w.r.t. $\rightarrowtriangle$), we introduce the notation $r' \overset{}{\multimap}\!\!\in r$ to combine these two possibilities.

Definition 1.37 ($\overset{\tau}{-}\!\!\in$) Let $\equiv_{\mathsf{n}}$ as in Definition 1.31. We define that $r' \overset{\tau}{-}\!\!\in r$ holds iff $r' \equiv_{\mathsf{n}} r$ with $\tau = \varepsilon$, or $r' \overset{\tau}{\rightarrowtriangle} r$.

Definition 1.38 (State Intersection) Let $s = (\langle fr_0, \ldots, fr_n \rangle, h, t, hp, sf, e, ic, \bot)$ and $s' = (\langle fr'_0, \ldots, fr'_{n'} \rangle, h', t', hp', sf', e', ic', \bot)$, where $fr_i = (pp_i, lv_i, os_i)$ and $fr'_i = (pp'_i, lv'_i, os'_i)$. Furthermore, let s and s' have disjoint sets of references (where only null and return addresses may be used in both states). Let $\equiv_{\mathsf{n}}, \rho, \sigma$ as defined in Definitions 1.31 and 1.34. Let $\textsc{References}(s)$ and $\textsc{References}(s')$, respectively, identify the references in the states.

We now define a function intersect: $\textsc{States} \times \textsc{States} \to \textsc{States} \cup \{※\}$. If the intersection as described below or according to Definition 1.33 does not exist, then $\text{intersect}(s, s') = ※$. Otherwise, $\text{intersect}(s, s') = \overline{s}$ where $\overline{s}$ is the state as described below.

Let $\overline{s} = (\langle \overline{fr_0}, \ldots, \overline{fr_n} \rangle, \overline{h}, \overline{t}, \overline{hp}, sf\sigma, \overline{e}, ic, \bot)$ where $\overline{fr_i} = (pp_i, lv_i\sigma, os_i\sigma)$. Using σ we define $\overline{e} = \bot$ if $e = \bot$, otherwise $\overline{e} = \sigma(e)$. The first four conditions of Definition 1.10 must hold as follows, otherwise $\overline{s}$ does not exist:

- $n = n'$ and $pp_i = pp'_i$ for all $0 \leq i \leq n$ (Definition 1.10(a))
- $e = \bot \Leftrightarrow e' = \bot$ (Definition 1.10(b))
- $ic = ic'$ (Definition 1.10(c))
- if $s'|_\pi$ is a return address, then $s|_\pi = s'|_\pi$ (Definition 1.10(d))
- if $s|_\pi$ is a return address, then $s'|_\pi = s|_\pi$ (Definition 1.10(d))

We now define the type component of the intersected state. Let $r \in \textsc{References}(s) \cup \textsc{References}(s')$ where $r = \mathsf{null}$ or the heap maps r to a value in $\textsc{Instances} \cup \textsc{Arrays}$:

$$\overline{t}(\sigma(r)) = \bigcap_{r' \in [r]_{\equiv_{\mathsf{n}}}} t_{r'}(r')$$

For $r \in \textsc{References}(s) \cup \textsc{References}(s')$ with $r \notin [\mathsf{null}]_{\equiv_\mathsf{n}}$ we now define the heap component:

$$\overline{h}(\sigma(r)) = \bigcap_{r' \in [r]_{\equiv_\mathsf{n}}} h_{r'}(r')$$

If for any reference r the intersection results in $\overline{h}(\sigma(r)) = \text{※}$, then $\overline{s}$ does not exist.

Finally, we define the heap predicates $\overline{hp}$. Let $r \neq r'$ be two references with $\overline{h}(\sigma(r)), \overline{h}(\sigma(r')) \in \textsc{Instances} \cup \textsc{Arrays}$:

(a) We add $\sigma(r)?$ if for all $r' \in [r]_{\equiv_\mathsf{n}}$ we have $r'?$.

(b) We add $\sigma(r) =^? \sigma(r')$ if we have $r_i \in [r]_{\equiv_\mathsf{n}}, r'_i \in [r']_{\equiv_\mathsf{n}}$ with $r_i =^? r'_i$, and for all $r_i \in [r]_{\equiv_\mathsf{n}}, r'_i \in [r']_{\equiv_\mathsf{n}}$ we either have $r_i =^? r'_i$ or r_i, r'_i are not in the same state. If we have $r_a \rightarrowtriangle [\pi]_{\equiv_\mathsf{n}}$ and $r_b \rightarrowtriangle [\pi']_{\equiv_\mathsf{n}}$ with $s_{r_a} = s_{r_b}$ we also demand that $r_a \curlyvee r_b$ exists. Furthermore, if $s_{r'}|_{\pi_a} = r_a \rightarrowtriangle [\pi]_{\equiv_\mathsf{n}}$, we need to have $s_{r'}|_{\pi_a} \curlyvee r'$. Similarly, if $s_r|_{\pi_b} = r_b \rightarrowtriangle [\pi']_{\equiv_\mathsf{n}}$, we need to have $s_r|_{\pi_b} \curlyvee r$.

(c) We add $\sigma(r) \curlyvee \sigma(r')$ if we have $r_i \mathrel{-\!\!\in} [r]_{\equiv_\mathsf{n}}$ and $r'_i \mathrel{-\!\!\in} [r']_{\equiv_\mathsf{n}}$ with $r_i \curlyvee r'_i$, and for all $r_i \mathrel{-\!\!\in} [r]_{\equiv_\mathsf{n}}$ and $r'_i \mathrel{-\!\!\in} [r']_{\equiv_\mathsf{n}}$ we either have $r_i \curlyvee r'_i$ or r_i, r'_i are not in the same state.

(d) We add $\sigma(r) \curlyvee \sigma(r)$ if we have $r' \mathrel{-\!\!\in} [r]_{\equiv_\mathsf{n}}$ with $r' \curlyvee r'$, and for all $r' \mathrel{-\!\!\in} [r]_{\equiv_\mathsf{n}}$ we have $r' \curlyvee r'$.

(e) We add $\sigma(r)\circlearrowleft_F$ with $F = \bigcup_i F_i$ if we have $r_i \mathrel{-\!\!\in} [r]_{\equiv_\mathsf{n}}$ with $r_i \circlearrowleft_{F_i}$, and we have $r_i \circlearrowleft_{F_i}$ for all $r_i \mathrel{-\!\!\in} [r]_{\equiv_\mathsf{n}}$.

(f) Let $[r]_{\equiv_\mathsf{n}} \neq [r']_{\equiv_\mathsf{n}}$ be any two equivalence classes of $\equiv_\mathsf{n}$. If

- for all $r_1, r_2 \in [r]_{\equiv_\mathsf{n}}$ we have $s_{r_1} = s_{r_2}$, and
- for all $r_1, r_2 \in [r']_{\equiv_\mathsf{n}}$ we have $s_{r_1} = s_{r_2}$, and
- for all $r_1 \in [r]_{\equiv_\mathsf{n}}, r_2 \in [r']_{\equiv_\mathsf{n}}$ we have $s_{r_1} \neq s_{r_2}$, and
- there are $\pi, \pi', r_1, r_2, \tau, \tau'$ with
  - $r_1 \in [r]_{\equiv_\mathsf{n}}$ with $r_1 \curlyvee s_r|_\pi$, and
  - $r_2 \in [r']_{\equiv_\mathsf{n}}$ with $r_2 \curlyvee s_{r'}|_{\pi'}$, and
  - $s_{r'}|_{\pi\tau} \in [r']_{\equiv_\mathsf{n}}$, and
  - $s_r|_{\pi'\tau'} \in [r]_{\equiv_\mathsf{n}}$

then we add $\rho([r]_{\equiv_\mathsf{n}}) =^? \rho([r']_{\equiv_\mathsf{n}})$ and $\rho([r]_{\equiv_\mathsf{n}}) \curlyvee \rho([r']_{\equiv_\mathsf{n}})$

The last item of adding heap predicates is different to the others, as here we add heap predicates derived from other heap predicates, instead of re-using existing heap predicates. The exact reason for this is rather technical (cf. the usages in the following proofs), but the main idea behind this is quite simple. In the intersected state we may have positions which do not exist in either of the two input states. In the case that we have two such positions where the corresponding original references (which, for example, provide type and existence information) are not in the same state, we neither have heap predicates nor explicit heap connections that can be used to describe possible connections in the intersected state. Thus, we need to consider the heap predicates which exist for corresponding predecessor references.

We now demonstrate Definition 1.38 using an example.

A:

$$\langle 23 \,|\, \mathsf{one}\colon r_1, \mathsf{two}\colon r_2 \,|\, \varepsilon \rangle$$

$$\begin{array}{c} r_1\colon \mathsf{List}(\mathsf{next}\colon r_3) \\ r_2\colon \mathsf{List}(\mathsf{next}\colon r_1) \\ r_3\colon \mathsf{List}(\mathsf{value}\colon i_1) \\ i_1\colon (-\infty, 1] \\ r_1 \;=^?\; r_2 \\ r_2 \;=^?\; r_3 \\ r_1 \;=^?\; r_3 \\ r_1 \searrow\!\!\!\swarrow r_1 \quad r_1 \searrow\!\!\!\swarrow r_2 \\ r_1 \searrow\!\!\!\swarrow r_3 \quad r_2 \searrow\!\!\!\swarrow r_2 \\ r_2 \searrow\!\!\!\swarrow r_3 \quad r_3 \searrow\!\!\!\swarrow r_3 \\ r_1 \circlearrowleft_{\emptyset} \; r_2 \circlearrowleft_{\emptyset} \; r_3 \circlearrowleft_{\emptyset} \end{array}$$

B:

$$\langle 23 \,|\, \mathsf{one}\colon r_4, \mathsf{two}\colon r_4 \,|\, \varepsilon \rangle$$

$$\begin{array}{c} r_4\colon \mathsf{List}(\mathsf{value}\colon i_2) \\ i_2\colon [0, \infty) \\ r_4 \searrow\!\!\!\swarrow r_4 \\ r_4 \circlearrowleft_{\{\mathsf{next}\}} \end{array}$$

C:

$$\langle 23 \,|\, \mathsf{one}\colon r_5, \mathsf{two}\colon r_5 \,|\, \varepsilon \rangle$$

$$\begin{array}{c} r_5\colon \mathsf{List}(\mathsf{value}\colon i_3, \mathsf{next}\colon r_5) \\ i_3\colon [0, 1] \\ r_5 \searrow\!\!\!\swarrow r_5 \\ r_5 \circlearrowleft_{\{\mathsf{next}\}} \end{array}$$

Figure 1.39.: States illustrating Definition 1.38

Example 1.31 Assume we intersect A and B shown in Fig. 1.39, resulting in state C. First, we compute the equivalence relation $\equiv_{\mathsf{n}}$, resulting in $r_1 \equiv_{\mathsf{n}} r_2 \equiv_{\mathsf{n}} r_3 \equiv_{\mathsf{n}} r_4$ and $i_1 \equiv_{\mathsf{n}} i_2$. We have $\rho([r_1]_{\equiv_{\mathsf{n}}}) = r_5$ and $\rho([i_1]_{\equiv_{\mathsf{n}}}) = i_3$.

This directly gives us the stack frame $\langle 23 | \mathsf{one}\colon r_5, \mathsf{two}\colon r_5 | \varepsilon \rangle$.

Next, we intersect values. For $[i_1]_{\equiv_{\mathsf{n}}}$ we intersect $h_A(i_1) = (-\infty, 1]$ and $h_B(i_2) = [0, \infty)$, which according to Definition 1.36 results in $[0, 1]$. Thus, in C we have $h_C(i_3) = [0, 1]$.

For $[r_1]_{\equiv_{\mathsf{n}}}$ we intersect four list objects. If we first consider the intersection of the objects referenced by r_1 and r_2, we obtain $\mathsf{List}(\mathsf{next}\colon r_5)$. Intersecting this intermediate result with the object referenced by r_3 gives $\mathsf{List}(\mathsf{next}\colon r_5, \mathsf{value}\colon i_3)$. Intersecting again with the object referenced by r_4 does not add new information.

Finally, we need to consider heap predicates. No reference is marked as $r?$, thus we do not need to add anything according to Definition 1.38(a). However, we have pairs of references marked using $=^?$ heap predicates. According to Definition 1.38(b) we would need to add $r_5 =^? r_5$. However, as this heap predicate is useless, we do not show it in C. Here it is important to understand that we not only need to have $r_i =^? r_j$ for all

$i \neq j$ with $i, j \in \{1, 2, 3\}$ to be forced to have a $=^?$ heap predicate in C. In addition we also require certain $\bigvee$ heap predicates for $\multimap$ predecessors of the references marked as $=^?$.

According to Definition 1.38(c,d) we need to add $r_5 \bigvee r_5$ to C. This is the case as we have $r_i \bigvee r_j$ with $r_i \in [r_j]_{\equiv_\mathsf{n}}$ for all $i, j \in \{1, 2, 3\}$.

In Definition 1.38(e) we see that the union of the field sets is $\{\mathsf{next}\}$. Thus, and as the conditions are met, we add $r_5 \circlearrowleft_{\{\mathsf{next}\}}$ to C.

As the condition of Definition 1.38(f) is not met, construction of C finishes with the state as shown in Fig. 1.39.

In order to show correctness of this intersection process (and, later, validity of equality refinement), we need to introduce some lemmas.

Since we retained field (or array contents) information even if only one out of many references in an equivalence class had this information, in the result we have all the positions that the two original states had. As explained above, there may also be positions which do not exist in neither of the input states.

Lemma 1.32 Let intersect$(s, s') = \overline{s} \neq ※$. Then $\text{SP}\textsc{os}(\overline{s}) \supseteq \text{SP}\textsc{os}(s) \cup \text{SP}\textsc{os}(s')$.

Proof. Let $s_r|_\pi = r$ with $s_r \in \{s, s'\}$. We show $\pi \in \text{SP}\textsc{os}(\overline{s})$ by induction over the length of π. Trivially, we have $\pi \in \text{SP}\textsc{os}(\overline{s})$ if $|\pi| = 1$. Otherwise, let $\pi = \pi'\tau$ with $\pi' \neq \varepsilon$ and $|\tau| = 1$. By the induction hypothesis, we also have $\pi' \in \text{SP}\textsc{os}(\overline{s})$.

- If $\tau = v \in \textsc{FieldIDs}$, we have $h_r(s_r|_{\pi'}) = f \in \textsc{Instances}$ with $v \in \text{dom}(f)$. According to Definition 1.36 in $\overline{s}$ we also have $\overline{h}(s|_{\pi'}) = \overline{f} \in \textsc{Instances}$ with $v \in \text{dom}(\overline{f})$. Thus, $\pi \in \text{SP}\textsc{os}(\overline{s})$.
- If $\tau = i \in \mathbb{N}$, we have $h_r(s_r|_{\pi'}) = (i_l, f) \in \textsc{Arrays}$ with $i \in \text{dom}(f)$. According to Definition 1.36 in $\overline{s}$ we also have $\overline{h}(s|_{\pi'}) = (\overline{i_l}, \overline{f}) \in \textsc{Arrays}$ with $i \in \text{dom}(\overline{f})$. Thus, $\pi \in \text{SP}\textsc{os}(\overline{s})$.
- If $\tau = \mathsf{len}$, we have $h_r(s_r|_{\pi'}) = (i_l, f) \in \textsc{Arrays}$. According to Definition 1.36 in $\overline{s}$ we also have $\overline{h}(s|_{\pi'}) = (\overline{i_l}, \overline{f}) \in \textsc{Arrays}$. Thus, $\pi \in \text{SP}\textsc{os}(\overline{s})$. ☐

Similar to the idea of validity of refinements, we need to make sure that all states represented by both of the two input states also is represented by the intersected state. Thus, if the intersected state does not exist, then there may be no concrete state represented by both of the input states.

Lemma 1.33 If $\text{intersect}(s, s') = ※$ then there is no concrete state c with $c \sqsubseteq s$ and $c \sqsubseteq s'$.

Proof. If a conflict is found according to Definition 1.33, we already showed the claim. Thus, the intersection may have failed due to a trivial reason (call stack height, different opcodes, exception, initialized classes, return addresses) or there is a reference $r \in \textsc{References}(s) \cup \textsc{References}(s')$ with $r \notin [\mathsf{null}]_{\equiv_\mathsf{n}}$ and $\bigcap_{r_i \in [r]_{\equiv_\mathsf{n}}} h_{r_i}(r_i) = ※$.

According to Definition 1.36 this can happen in any of the following cases. Let $r \equiv_\mathsf{n} r'$ where $r \neq r'$ and $r \not\equiv_\mathsf{n} \mathsf{null} \not\equiv_\mathsf{n} r'$. Let π, π' be positions with $s_r|_\pi = r$ and $s_{r'}|_{\pi'} = r'$.

- $h_r(r) = V_r \in \textsc{Integers}, h_{r'}(r') = V_{r'} \in \textsc{Integers}, V_r \cap V_{r'} = \emptyset$. With Lemma 1.27 we have $c|_\pi = c|_{\pi'}$ with $h_c(c|_\pi) = \{z\} \in \textsc{Integers}$. If $c \sqsubseteq s$ and $c \sqsubseteq s'$, we had $\{z\} \subseteq V_r$ and $\{z\} \subseteq V_{r'}$, thus $z \in V_r \cap V_{r'}$. As this is not the case, the claim follows.

- $h_r(r) = V_r \in \textsc{Floats}, h_{r'}(r') = V_{r'} \in \textsc{Floats}, V_r \neq V_{r'}$ and $V_r \neq \bot \neq V_{r'}$. With Lemma 1.27 we have $c|_\pi = c|_{\pi'}$ with $h_c(c|_\pi) = z \in \textsc{Floats}$ with $z \neq \bot$. If $c \sqsubseteq s$ and $c \sqsubseteq s'$, we had $V_r, V_{r'} \in \{z, \bot\} \subseteq \textsc{Floats}$. As this is not the case, the claim follows.

- $h_r(r) = f \in \textsc{Instances}, \text{dom}(f) \neq \emptyset, h_{r'}(r') = (i_{l,r'}, f_{r'}) \in \textsc{Arrays}$. With Lemma 1.27 we have $c|_\pi = c|_{\pi'}$. Due to Definition 1.10(i,j) we cannot have $c \sqsubseteq s$ and $c \sqsubseteq s'$, as $\text{dom}(f) \neq \emptyset$.

- $h_{r'}(r') = f' \in \textsc{Instances}, \text{dom}(f') \neq \emptyset, h_r(r) = (i_{l,r}, f_r) \in \textsc{Arrays}$. With Lemma 1.27 we have $c|_\pi = c|_{\pi'}$. Due to Definition 1.10(i,j) we cannot have $c \sqsubseteq s$ and $c \sqsubseteq s'$, as $\text{dom}(f') \neq \emptyset$. □

As there may be positions in the intersected state that do not exist in any of the input states, for the upcoming proofs we need a way to access the corresponding equivalence class. For this, we introduce the notation of $[\pi]_{\equiv_\mathsf{n}}$ denoting the equivalence class corresponding to a position π in the intersected state.

Definition 1.40 ($[\pi]_{\equiv_\mathsf{n}}$) For $\overline{s} = \text{intersect}(s, s')$ with $\overline{s} \neq ※$, $\pi \in \textsc{SPos}(\overline{s})$, and $\equiv_\mathsf{n}$ as in Definition 1.31 we define $[\pi]_{\equiv_\mathsf{n}} := [r]_{\equiv_\mathsf{n}}$ with $\rho([r]_{\equiv_\mathsf{n}}) = \overline{s}|_\pi$.

The following lemma helps us to reason about information in $\overline{s}$ that may be at a position which does not exist in any of the input states.

Lemma 1.34 Let $※ \neq \overline{s} = \text{intersect}(s, s')$. Let c be a concrete state with $c \sqsubseteq s$ and $c \sqsubseteq s'$. Let $\pi \in \text{SPos}(\overline{s})$. Then for all π' with $r \in [\pi]_{\equiv_\mathsf{n}}$ and $s_r|_{\pi'} = r$ we have $c|_\pi = c|_{\pi'}$.

Proof. Let $r \in [\pi]_{\equiv_\mathsf{n}}$, i.e., $\sigma(r) = \overline{s}|_\pi$ and $s_r|_{\pi'} = r$. Thus, we also have $\overline{s}|_{\pi'} = \overline{s}|_\pi$ and, hence, $\overline{s}|_{\pi'} \equiv_\mathsf{n} \overline{s}|_\pi$. With Lemma 1.27 we have $c|_{\pi'} = c|_\pi$. □

The following theorem states that intersection is correct in the sense that each concrete state represented by both of the input states also is represented by the intersected state.

Theorem 1.35 Let $s, s' \in \text{States}$. If $\text{intersect}(s, s') = \overline{s} \neq ※$, for each concrete state c with $c \sqsubseteq s$ and $c \sqsubseteq s'$ we have $c \sqsubseteq \overline{s}$.

Proof. Let $\text{intersect}(s, s') = \overline{s} \neq ※$ where $s = (\langle fr_0, \ldots, fr_n\rangle, h, t, hp, sf, e, ic, \bot)$, $s' = (\langle fr'_0, \ldots, fr'_n\rangle, h', t', hp', sf', e', ic', \bot)$, $\overline{s} = (\langle \overline{fr_0}, \ldots, \overline{fr_n}\rangle, \overline{h}, \overline{t}, \overline{hp}, \overline{sf}, \overline{e}, ic, \bot)$, $fr_i = (pp_i, lv_i, os_i)$, $fr'_i = (pp_i, lv'_i, os'_i)$, and $\overline{fr_i} = (pp_i, \overline{lv_i}, \overline{os_i})$. Let s, s' be states with disjoint sets of references (where only null and return addresses may be used in both states). Let c be a concrete state with $c \sqsubseteq s$ and $c \sqsubseteq s'$.

We show $c \sqsubseteq \overline{s}$ by proving the individual items of Definition 1.10. Let $\pi, \pi' \in \text{SPos}(c)$.

(a – c) Trivial.

(d) The claim directly follows from Definitions 1.34 and 1.38.

(e) Let $h_c(c|_\pi) = z \in \text{Floats}$ with $z \neq \bot$. Assume $\pi \in \text{SPos}(\overline{s})$. With Lemma 1.34 we have $c|_\pi = c|_{\pi'}$ for all π' with $s_r|_{\pi'} = r \in [\pi]_{\equiv_\mathsf{n}}$. With Definition 1.10(e), for each $r \in [\pi]_{\equiv_\mathsf{n}}$ we have $h_r(r) \in \{z, \bot\} \subseteq \text{Floats}$. Thus, with Definition 1.36 we then also have $\overline{h}(\overline{s}|_\pi) \in \{z, \bot\}$.

(f) Let $h_c(c|_\pi) = \{z\} \in \text{Integers}$. Assume $\pi \in \text{SPos}(\overline{s})$. With Definition 1.36 we have $\overline{h}(\overline{s}|_\pi) = \bigcap_{r \in [\pi]_{\equiv_\mathsf{n}}} h_r(r)$. With Lemma 1.34, we also know that for all r with $s_r|_{\pi'} = r$ and $r \in [\pi]_{\equiv_\mathsf{n}}$ we have $c|_\pi = c|_{\pi'}$. Thus, with Definition 1.10(f) we also have $\{z\} \subseteq h_r(r)$ for all $r \in [\pi]_{\equiv_\mathsf{n}}$. Hence, the claim follows.

(g) Let $t_c(c|_\pi) = \{T\} \in 2^{\mathbb{N} \times (\text{PrimTypes} \cup \text{ClassNames})}$. Assume $\pi \in \text{SPos}(\overline{s})$. We have $\overline{h}(\overline{s}|_\pi) = \bigcap_{r \in [\pi]_{\equiv_\mathsf{n}}} t_r(r)$. With Lemma 1.34 we also know that for each π' with $s_r|_{\pi'} = r$ we have $c|_\pi = c|_{\pi'}$. Thus, with Definition 1.10(g) we also have $\{T\} \subseteq t_r(r)$ for all $r \in [\pi]_{\equiv_\mathsf{n}}$. Hence, the claim follows.

(h) We have $h_c(c|_\pi) = \mathsf{null}$. Assume $\pi \in \text{SPos}(\overline{s})$. If $[\pi]_{\equiv_\mathsf{n}} = [\mathsf{null}]_{\equiv_\mathsf{n}}$ then $\overline{s}|_\pi = \mathsf{null}$. Otherwise, we have $\overline{s}|_\pi = r$ for some reference $r \neq \mathsf{null}$. We need to show that $r?$ and $\overline{h}(r) = f \in \text{Instances}$ with $\text{dom}(f) = \emptyset$.

Because, with Lemma 1.34, we have $c|_\pi = c|_{\pi'}$ for all r', π' with $s_{r'}|_{\pi'} = r'$ and $r' \in [\pi]_{\equiv_\mathsf{n}}$, we also have $r'?$ and $h_{r'}(r') = f_{r'} \in \text{Instances}$ with $\text{dom}(f_{r'}) = \emptyset$ (with Definition 1.10(h) and $r' \notin [\mathsf{null}]_{\equiv_\mathsf{n}}$). Thus, with Definition 1.38(a) we also have $r?$. Furthermore, with Definition 1.36 we also have $\overline{h}(r) = f \in \text{Instances}$ with $\text{dom}(f) = \emptyset$.

(i) Let $h_c(c|_\pi) = f_c \in \text{Instances}$. Assume $\pi \in \text{SPos}(\overline{s})$. With Lemma 1.34 we know that for each π' with $s_r|_{\pi'} = r$ and $r \in [\pi]_{\equiv_\mathsf{n}}$ we have $c|_\pi = c|_{\pi'}$. Thus, with Definition 1.10(i) we also have $h_r(r) \in \text{Instances}$ and $\text{dom}(f_c) \supseteq \text{dom}(h_r(r))$ for all $r \in [\pi]_{\equiv_\mathsf{n}}$. With Definition 1.36 we have $\overline{h}(\overline{s}|_\pi) = \bigcup_{r \in [\pi]_{\equiv_\mathsf{n}}} h_r(r)$ (with $f \cup f'$ as defined in Definition 1.35). Hence, the claim follows.

(j) Let $h_c(c|_\pi) = (i_{l,c}, f_c) \in \text{Arrays}$. Assume $\pi \in \text{SPos}(\overline{s})$. With Lemma 1.34 we know that for each π' with $s_r|_{\pi'} = r$ and $r \in [\pi]_{\equiv_\mathsf{n}}$ we have $c|_\pi = c|_{\pi'}$. For all such r with $h_r(r) = f_r \in \text{Instances}$ with Definition 1.10(j) we know that $\text{dom}(f_r) = \emptyset$. Similarly, if $h_r(r) = (i_{l,r}, f_r) \in \text{Arrays}$ we have $\text{dom}(f_c) \supseteq \text{dom}(f_r)$. With Definition 1.36 we then either have

- $\overline{h}(\overline{s}|_\pi) = (i_l, f) \in \text{Arrays}$ where $f = \bigcup\{f_r \mid r \in [\pi]_{\equiv_\mathsf{n}} \wedge (i_{l,r}, f_r) = h_r(r)\}$, or
- $\overline{h}(\overline{s}|_\pi) = f \in \text{Instances}$ where $f = \bigcup\{f_r \mid r \in [\pi]_{\equiv_\mathsf{n}} \wedge f_r = h_r(r)\}$ with $\text{dom}(f) = \emptyset$.

Hence, the claim follows.

(k) We have $c|_\pi \neq c|_{\pi'}$ and $\pi, \pi' \in \text{SPos}(\overline{s})$. Assume we had $\overline{s}|_\pi = \overline{s}|_{\pi'}$. Then, with Lemma 1.34 consider $\widetilde{\pi}, \widetilde{\pi}'$ and r, r' with $s_r|_{\widetilde{\pi}} = r$, $s_{r'}|_{\widetilde{\pi}'} = r'$, $r \in [\pi]_{\equiv_\mathsf{n}}, r' \in [\pi']_{\equiv_\mathsf{n}}$ so that we have $c|_{\widetilde{\pi}} = c|_\pi$ and $c|_{\widetilde{\pi}'} = c|_{\pi'}$. With Lemma 1.34 we then also had $c|_\pi = c|_{\widetilde{\pi}} = c|_{\widetilde{\pi}'} = c|_{\pi'}$. This contradicts the hypothesis and we conclude $\overline{s}|_\pi \neq \overline{s}|_{\pi'}$.

(l) We have $c|_\pi = c|_{\pi'}$ with $\pi \neq \pi'$, $h_c(c|_\pi) \in \text{Instances} \cup \text{Arrays}$, and $\pi, \pi' \in \text{SPos}(\overline{s})$. Thus, we need to show that $\overline{s}|_\pi = \overline{s}|_{\pi'}$ or $\overline{s}|_\pi =^? \overline{s}|_{\pi'}$. If $[\pi]_{\equiv_\mathsf{n}} = [\pi']_{\equiv_\mathsf{n}}$ we have $\overline{s}|_\pi = \overline{s}|_{\pi'}$. Thus, we need to consider the case that $[\pi]_{\equiv_\mathsf{n}} \neq [\pi']_{\equiv_\mathsf{n}}$.

Let $\widetilde{\pi}, \widetilde{\pi}'$ with $r = s_r|_{\widetilde{\pi}} \in [\pi]_{\equiv_\mathsf{n}}$ and $r' = s_{r'}|_{\widetilde{\pi}'} \in [\pi']_{\equiv_\mathsf{n}}$. As $[\pi]_{\equiv_\mathsf{n}} \neq [\pi']_{\equiv_\mathsf{n}}$ and we ensured that the references in the states are disjoint we have $r \neq r'$. With $s_r|_{\widetilde{\pi}} \in [\pi]_{\equiv_\mathsf{n}}$ and Lemma 1.34 we have $c|_{\widetilde{\pi}} = c|_\pi$. Similarly, we have $c|_{\widetilde{\pi}'} = c|_{\pi'}$. Thus, if $s_r = s_{r'}$ with Definition 1.10(l) we have $s_r|_{\widetilde{\pi}} =^? s_{r'}|_{\widetilde{\pi}'}$.

Assume we have $s_{r_a}|_{\pi_a} = r_a \xrightarrow{\tau_a} [\pi]_{\equiv_\mathsf{n}}$ and $s_{r_b}|_{\pi_b} = r_b \xrightarrow{\tau_b} [\pi']_{\equiv_\mathsf{n}}$. With Lemma 1.24 we have $c|_{\pi_a\tau_a} = c|_\pi = c|_{\pi'} = c|_{\pi_b\tau_b}$. We also know $\pi_a\tau'_a \notin \text{SPos}(s_{r_a})$ and $\pi_b\tau'_b \notin$

$\text{SPos}(s_{r_b})$ for $\varepsilon \neq \tau'_a \trianglelefteq \tau_a$ and $\varepsilon \neq \tau'_b \trianglelefteq \tau_b$. Thus, with Definition 1.10(m) we have $s_{r_a}|_{\pi_a} \curlyvee s_{r_b}|_{\pi_b}$ if $s_{r_a} = s_{r_b}$.

Now assume we have $s_{r'}|_{\pi_a} = r_a \xrightarrow{\tau_a} [\pi]_{\equiv_\mathsf{n}}$. With Lemma 1.24 we have $c|_{\pi_a\tau_a} = c|_\pi = c|_{\pi'}$. Thus, with Definition 1.10(m) we have $s_{r'}|_{\pi_a} \curlyvee r'$. Similarly, we also have $s_r|_{\pi_b} \curlyvee r$ for $s_r|_{\pi_b} = r_b \rightarrow [r']_{\equiv_\mathsf{n}}$.

According to Definition 1.38(b) we then also have $\overline{s}|_\pi =^? \overline{s}|_{\pi'}$.

If there are no $\widetilde{\pi}, \widetilde{\pi}'$ with $s_r = s_{r'}$ we know that for all $\hat{r} \in [\pi]_{\equiv_\mathsf{n}}$ we have $s_{\hat{r}} = s_r \neq s_{r'}$, $\pi' \notin \text{SPos}(s_r)$, and with $c|_{\widetilde{\pi}} = c|_\pi = c|_{\pi'}$ and Definition 1.10(m) we have $s_r \curlyvee s_r|_{\overline{\pi'}}$. Similarly, for all $\hat{r}' \in [\pi']_{\equiv_\mathsf{n}}$ we have $s_{\hat{r}'} = s_{r'} \neq s_r$, $\pi \notin \text{SPos}(s_{r'})$, and with $c|_\pi = c|_{\pi'} = c|_{\widetilde{\pi}'}$ and Definition 1.10(m) we have $s_{r'} \curlyvee s_r|_{\overline{\pi}}$. Thus, according to Definition 1.38(f) we also have $\overline{s}|_\pi =^? \overline{s}_{\pi'}$.

(m) We have $c|_\pi = c|_{\pi'}$ with $\pi \neq \pi'$ and $h_c(c|_\pi) \in \text{Instances} \cup \text{Arrays}$. We also have $\{\pi, \pi'\} \not\subseteq \text{SPos}(\overline{s})$, so that we need to show $\overline{s}|_{\overline{\pi}} \curlyvee \overline{s}|_{\overline{\pi'}}$, or $\overline{s}|_{\overline{\pi}} = \overline{s}|_{\overline{\pi'}}$ where π, π' have the same suffix w.r.t. $\overline{s}$.

Let $\widetilde{\pi}, \widetilde{\pi}'$ with $r = s_r|_{\widetilde{\pi}} \twoheadleftarrow [\overline{\pi}_{\overline{s}}]_{\equiv_\mathsf{n}}$ and $r' = s_{r'}|_{\widetilde{\pi}'} \twoheadleftarrow [\overline{\pi'}_{\overline{s}}]_{\equiv_\mathsf{n}}$. With Lemmas 1.24 and 1.34 we have $c|_{\widetilde{\pi}\tilde{\tau}} = c|_{\overline{\pi}_{\overline{s}}}$ and $c|_{\widetilde{\pi}'\tilde{\tau}'} = c|_{\overline{\pi'}_{\overline{s}}}$. Thus, we have τ, τ' with $c|_{\widetilde{\pi}\tilde{\tau}\tau} = c|_{\widetilde{\pi}'\tilde{\tau}'\tau'} = c|_{\overline{\pi}_{\overline{s}}\tau} = c|_{\overline{\pi'}_{\overline{s}}\tau'} = c|_\pi = c|_{\pi'}$.

If $s_r = s_{r'}$, with Definition 1.10(m) we have $s_r|_{\widetilde{\pi}} \curlyvee s_r|_{\widetilde{\pi}'}$, or $s_r|_{\widetilde{\pi}} = s_r|_{\widetilde{\pi}'}$ where $\widetilde{\pi}\tau$, $\widetilde{\pi}'\tau'$ have the same suffix w.r.t. s_r. In the latter case, if the joins heap predicate $s_r|_{\widetilde{\pi}} \curlyvee s_r|_{\widetilde{\pi}'}$ is missing, we have $\tilde{\tau}\tau = \tilde{\tau}'\tau'$ and from $s_r|_{\widetilde{\pi}} = s_r|_{\widetilde{\pi}'}$ we conclude $s_r|_{\widetilde{\pi}} \equiv_\mathsf{n} s_r|_{\widetilde{\pi}'}$, thus $\overline{s}|_{\overline{\pi}} \equiv_\mathsf{n} \overline{s}|_{\overline{\pi'}}$. With this, we also have $\overline{s}|_{\overline{\pi}} = \overline{s}|_{\overline{\pi'}}$. Otherwise, the joins heap predicate $s_r|_{\widetilde{\pi}} \curlyvee s_r|_{\widetilde{\pi}'}$ exists for all $\widetilde{\pi}, \widetilde{\pi}'$. With Definition 1.38(c) we also have $\overline{s}|_{\overline{\pi}} \curlyvee \overline{s}|_{\overline{\pi'}}$.

If there are no $\widetilde{\pi}, \widetilde{\pi}'$ with $s_r = s_{r'}$ we know that for all $\hat{r} \in [\overline{\pi}_{\overline{s}}]_{\equiv_\mathsf{n}}$ we have $s_{\hat{r}} = s_r \neq s_{r'}$, $\overline{\pi'}_{\overline{s}} \notin \text{SPos}(s_r)$, and with $c|_{\widetilde{\pi}\tau} = c|_\pi = c|_{\pi'}$ and Definition 1.10(m) we have $s_r \curlyvee s_r|_{\overline{\pi'}}$. Similarly, for all $\hat{r}' \in [\overline{\pi'}_{\overline{s}}]_{\equiv_\mathsf{n}}$ we have $s_{\hat{r}'} = s_{r'} \neq s_r$, $\overline{\pi}_{\overline{s}} \notin \text{SPos}(s_{r'})$, and with $c|_\pi = c|_{\pi'} = c|_{\widetilde{\pi}'\tau'}$ and Definition 1.10(m) we have $s_{r'} \curlyvee s_r|_{\overline{\pi}}$. Thus, according to Definition 1.38(f) we also have $\overline{s}|_{\overline{\pi}} \curlyvee \overline{s}_{\overline{\pi'}}$.

Thus, the claim follows.

(n) Assume $\pi = \alpha\tau$ and $\pi' = \alpha\tau'$ where $\tau \neq \varepsilon$ and τ, τ' do not have a common intermediate reference from α in c, and we have $c|_\pi = c|_{\pi'}$ with $c|_\pi \in \text{Instances} \cup \text{Arrays}$. If $\pi, \pi' \in \text{SPos}(\overline{s})$, it suffices to show that $\overline{s}|_\pi = \overline{s}|_{\pi'}$. If this does not hold, we need to show $\overline{s}|_{\overline{\alpha}} \curlyvee \overline{s}|_{\overline{\alpha}}$ and, if $\tau' = \varepsilon$, also $\overline{s}|_{\overline{\alpha}} \circlearrowleft_F$ with $F \subseteq \tau$.

- First consider the case that $\{\alpha\tau, \alpha\tau'\} \subseteq \text{SPos}(\overline{s})$. If $\overline{s}|_{\alpha\tau} = \overline{s}|_{\alpha\tau'}$, the claim follows. Otherwise, let $\widetilde{\pi}, \widetilde{\pi}', \widetilde{\alpha}$ with $r = s_r|_{\widetilde{\pi}} \in [\pi]_{\equiv_\mathsf{n}}$, $r' = s_{r'}|_{\widetilde{\pi}'} \in [\pi']_{\equiv_\mathsf{n}}$, and $r_\alpha = s_{r_\alpha}|_{\widetilde{\alpha}} \in [\alpha]_{\equiv_\mathsf{n}}$.

- First assume we have $s_r = s_{r'} = s_{r_\alpha}$. With $\overline{s}|_\pi \neq \overline{s}|_{\pi'}$ we have $[\pi]_{\equiv_\mathsf{n}} \neq [\pi']_{\equiv_\mathsf{n}}$, thus we also have $s_r|_{\widetilde{\pi}} \neq s_r|_{\widetilde{\pi}'}$. With Lemma 1.34 we have $c|_\pi = c|_{\widetilde{\pi}}$, $c|_{\pi'} = c|_{\widetilde{\pi}'}$, and $c|_\alpha = c|_{\widetilde{\alpha}}$. Furthermore, we have $c|_{\widetilde{\alpha}\tau} = c|_{\widetilde{\pi}} = c|_\pi = c|_{\pi'} = c|_{\widetilde{\pi}'} = c|_{\widetilde{\alpha}\tau'}$. Thus, with Definition 1.10(n) we also have $s_r|_{\widetilde{\alpha}} \bigvee\!\!\!\!\!/\, s_r|_{\widetilde{\alpha}}$. If $\tau' = \varepsilon$, with Definition 1.10(n) we also have $s_r|_{\widetilde{\alpha}} \circlearrowleft_F$ with $F \subseteq \tau$.
- Now assume there are no $\widetilde{\pi}, \widetilde{\pi}', \widetilde{\alpha}$ with $r = s_r|_{\widetilde{\pi}} \in [\pi]_{\equiv_\mathsf{n}}, r' = s_{r'}|_{\widetilde{\pi}'} \in [\pi']_{\equiv_\mathsf{n}}, r_\alpha = s_{r_\alpha}|_{\widetilde{\alpha}} \in [\alpha]_{\equiv_\mathsf{n}}$, and $s_r = s_{r'} = s_{r_\alpha}$. With Lemma 1.34 we have $c|_\alpha = c|_{\widetilde{\alpha}}$, thus we also have $c|_{\widetilde{\alpha}\tau} = c|_{\widetilde{\alpha}\tau'}$. If we do not have $s_{r_\alpha}|_{\widetilde{\alpha}} \bigvee\!\!\!\!\!/\, s_{r_\alpha}|_{\widetilde{\alpha}}$, with Definition 1.10(n) we know that $s_{r_\alpha}|_{\widetilde{\alpha}\tau} = s_{r_\alpha}|_{\widetilde{\alpha}\tau'}$. With $\widetilde{\alpha}\tau = \widetilde{\pi}$ and $\widetilde{\alpha}\tau' = \widetilde{\pi}'$ this contradicts the assumption stated above. As a consequence, we have $s_{r_\alpha}|_{\widetilde{\alpha}} \bigvee\!\!\!\!\!/\, s_{r_\alpha}|_{\widetilde{\alpha}}$. Similarly, if $\tau = \varepsilon$ we also have $s_{r_\alpha}|_{\widetilde{\alpha}} \circlearrowleft_F$ with $F \subseteq \tau$.

If we have $s_{\hat{r}}|_{\widehat{\pi}} = \hat{r} \xrightarrow{\hat{\tau}} [\alpha]_{\equiv_\mathsf{n}}$, with Lemma 1.24 we also have $c|_{\widehat{\pi}\hat{\tau}\tau} = c|_{\alpha\tau} = c|_{\alpha\tau'} = c|_{\widehat{\pi}\hat{\tau}\tau'}$ where $\overline{\widehat{\pi}\hat{\tau}}_{s_{\hat{r}}} = \widehat{\pi}$. Thus, with Definition 1.10(n) we also have $s_{\hat{r}}|_{\widehat{\pi}} \bigvee\!\!\!\!\!/\, s_{\hat{r}}|_{\widehat{\pi}}$.

As we have $s_{r_\alpha}|_{\widetilde{\alpha}} \bigvee\!\!\!\!\!/\, s_{r_\alpha}|_{\widetilde{\alpha}}$ for all $s_{r_\alpha}|_{\widetilde{\alpha}} \in [\alpha]_{\equiv_\mathsf{n}}$, according to Definition 1.38(d) we have $\overline{s}|_\alpha \bigvee\!\!\!\!\!/\, \overline{s}|_\alpha$. If $\tau = \varepsilon$, we also have $s_{r_\alpha}|_{\widetilde{\alpha}} \circlearrowleft_{F_i}$ with $F_i \subseteq \tau$ for all $s_{r_\alpha}|_{\widetilde{\alpha}} \in [\alpha]_{\equiv_\mathsf{n}}$. Let $s_r|_{\pi''} \xrightarrow{\tau''} r_\alpha$. With Lemma 1.24 we have $c|_{\pi''\tau''} = c|_\alpha$ with $\varepsilon \neq \tau''_1 \trianglelefteq \tau''$ and $\pi''\tau''_1 \notin \mathrm{SPos}(s_r)$. Thus, with Definition 1.10(o) we also have $s_r|_{\pi''} \circlearrowleft_{F''}$ with $F'' \subseteq \tau$. With Definition 1.38(e) we then also have $\overline{s}|_\alpha \circlearrowleft_F$ with $F \subseteq \tau$.

- Now assume we have $\{\alpha\tau, \alpha\tau'\} \not\subseteq \mathrm{SPos}(\overline{s})$. Consider any position $\widetilde{\alpha}$ with $r_\alpha = s_{r_\alpha}|_{\widetilde{\alpha}}$ and $r_\alpha \in [\overline{\alpha}_{\overline{s}}]_{\equiv_\mathsf{n}}$. Let β with $\alpha = \overline{\alpha}_{\overline{s}}\beta$. By construction we know $\overline{\widetilde{\alpha}\beta}_{s_{r_\alpha}} = \widetilde{\alpha}$. According to Lemma 1.34 we have $c|_{\overline{\alpha}_{\overline{s}}} = c|_{\widetilde{\alpha}}$. Thus, we also have $c|_{\overline{\alpha}_{\overline{s}}\beta\tau} = c|_{\alpha\tau} = c|_{\alpha\tau'} = c|_{\widetilde{\alpha}\beta\tau} = c|_{\widetilde{\alpha}\beta\tau'}$. As we also have $\{\widetilde{\alpha}\tau, \widetilde{\alpha}\tau'\} \not\subseteq \mathrm{SPos}(\overline{s})$, with Definition 1.10(n) we then also have $s_{r_\alpha}|_{\widetilde{\alpha}} \bigvee\!\!\!\!\!/\, s_{r_\alpha}|_{\widetilde{\alpha}}$. If we have $s_{\hat{r}}|_{\widehat{\pi}} = \hat{r} \xrightarrow{\hat{\tau}} [\alpha]_{\equiv_\mathsf{n}}$, with Lemma 1.24 we also have $c|_{\widehat{\pi}\hat{\tau}\tau} = c|_{\alpha\tau} = c|_{\alpha\tau'} = c|_{\widehat{\pi}\hat{\tau}\tau'}$ where $\overline{\widehat{\pi}\hat{\tau}}_{s_{\hat{r}}} = \widehat{\pi}$. Thus, with Definition 1.10(n) we also have $s_{\hat{r}}|_{\widehat{\pi}} \bigvee\!\!\!\!\!/\, s_{\hat{r}}|_{\widehat{\pi}}$.

  Thus, according to Definition 1.38(d) we also have $\overline{s}|_{\widetilde{\alpha}} \bigvee\!\!\!\!\!/\, \overline{s}|_{\widetilde{\alpha}}$. If $\tau = \varepsilon$, we also have $s_{r_\alpha}|_{\widetilde{\alpha}} \circlearrowleft_F$ with $F \subseteq \tau$. Let $s_r|_{\pi''} \xrightarrow{\tau''} r_\alpha$. With Lemma 1.24 we have $c|_{\pi''\tau''} = c|_\alpha$ with $\varepsilon \neq \tau''_1 \trianglelefteq \tau''$ and $\pi''\tau''_1 \notin \mathrm{SPos}(s_r)$. Thus, with Definition 1.10(o) we also have $s_r|_{\pi''} \circlearrowleft_{F''}$ with $F'' \subseteq \tau$. With Definition 1.38(e) we then also have $\overline{s}|_{\widetilde{\alpha}} \circlearrowleft_{F'}$ with $F \subseteq \tau$. □

(o – r) Not applicable, as c is concrete.

1.5.5. Validity of Equality Refinement

The intersection process introduced so far only dealt with arbitrary input states. However, in the equality refinement we use states $s[r/r']$ and $s[r'/r]$. Thus, we first need to show the relationship of these states and s.

The main complication is that when we replace references in a state, say r by r', then the replaced reference (r) may occur somewhere in the heap reachable from the replacement reference (r'). This corresponds to a cycle in the heap. By making use of the fact that the shape of the heap is rather simple even in the presence of cycles when considering concrete states, we can identify positions in $s[r/r']$ and corresponding positions in s which lead to the same reference in both states.

Lemma 1.36 Let s be a state with $r =^? r'$ where $h(r) \in \textsc{Instances} \cup \textsc{Arrays}$, $h(r') \in \textsc{Instances} \cup \textsc{Arrays}$. Let $\Pi = \{\pi \mid s|_\pi = r\}, \Pi' = \{\pi' \mid s|_{\pi'} = r'\}$. Let c be a concrete state with $c \sqsubseteq s$ and $c|_\pi = c|_{\pi'}$ for all $\pi \in \Pi, \pi' \in \Pi'$. Let $s' = s[r/r']$.

Then for any position $\widehat{\pi}$ there is a position β with $s|_\beta = s'|_{\widehat{\pi}}$ and $c|_\beta = c|_{\widehat{\pi}}$.

Proof. Let $\mathcal{N} = \{\pi\tau \in \textsc{SPos}(s') \mid \pi \in \Pi\}$ be the positions where s may differ from s'. For $\widehat{\pi} \notin \mathcal{N}$ the proof is trivial. Thus, we only show the claim for $\widehat{\pi} := \pi\tau \in \mathcal{N}$ with $\pi \in \Pi$.

- If $\Pi' \cap \mathcal{N} = \emptyset$, we have $r' = s|_{\pi'} = s'|_\pi$ for all $\pi \in \Pi, \pi' \in \Pi'$. We also have $s'|_{\pi\tau} = s|_{\pi'\tau}$ and $c|_{\pi\tau} = c|_{\pi'\tau}$. Thus, with $\beta = \pi'\tau$ for $\pi' \in \Pi'$ the claim follows.
- Otherwise, let $\pi\alpha \in \Pi' \cap \mathcal{N}$ with $s|_\pi = r, s|_{\pi\alpha} = r'$ ($\alpha \neq \varepsilon$). In other words, α is a path leading from r to r' in s. Then we have $r' = s|_{\pi\alpha} = s'|_\pi$. However, we do not necessarily have $s|_{\pi\alpha\tau} = s'|_{\pi\tau}$, as for example $s|_{\pi\alpha\tau} = r$ and $s'|_{\pi\tau} = r'$ is possible (because r is replaced with r' in s').
  - Assume $s|_{\pi\alpha\tau} = s'|_{\pi\tau}$. Then with $c|_\pi = c|_{\pi'}$ for all $\pi \in \Pi$, $\pi' \in \Pi'$ we also have $c|_{\pi\alpha\tau} = c|_{\pi\tau}$. Thus, with $\beta = \pi\alpha\tau$ the claim follows.
  - Assume $s|_{\pi\alpha\tau} \neq s'|_{\pi\tau}$. Thus, we know that r is reached from r' in s. We split τ into $\tau = \tau_1\tau_2$ with
    * $s|_{\pi\alpha\tau_1} = r$
    * for all $\varepsilon \neq \tau_2' \trianglelefteq \tau_2$ we have $s|_{\pi\alpha\tau_1\tau_2'} \neq r$ ($\tau_2 = \varepsilon$ is allowed)

    This means r does not appear along τ_2. As the path from r' to r is traversed at least once, this is represented in τ_1. Furthermore, if the path contains multiple traversals along the cycle $r' \to r \to r' \to \cdots$, these traversals are part of τ_1.

Then we have $s|_{\pi\alpha\tau_2} = s'|_{\pi\tau_1\tau_2} = s'|_{\pi\tau}$. With $c|_\pi = c|_{\pi'}$ for all $\pi \in \Pi, \pi' \in \Pi'$ we also know $c|_{\pi\alpha\tau} = c|_{\pi\tau}$. Thus, with $\beta = \pi\alpha\tau_2$ the claim follows. □

Lemma 1.37 Let s be a state with $r =^? r'$ where $h(r) \in \textsc{Instances} \cup \textsc{Arrays}$, $h(r') \in \textsc{Instances} \cup \textsc{Arrays}$. Let $\Pi = \{\pi \mid s|_\pi = r\}, \Pi' = \{\pi' \mid s|_{\pi'} = r'\}$. Then, for every concrete state c with $c \sqsubseteq s$ and $c|_\pi = c|_{\pi'}$ for any $\pi \in \Pi, \pi' \in \Pi'$ we have $c \sqsubseteq s[r/r']$ and $c \sqsubseteq s[r'/r]$.

Proof. W.l.o.g. we just prove $c \sqsubseteq s'$ where $s' = s[r/r']$. For any position $\pi'\tau \in \textsc{SPos}(s)$ with $\pi' \in \Pi'$ we have $s'|_{\pi\tau} = s'|_{\pi'\tau}$ for all $\pi \in \Pi$. Let $\mathcal{N}$ be defined as in Lemma 1.36.

We show the claim by proving the individual items of Definition 1.10. Let $\pi \in \textsc{SPos}(c)$. W.l.o.g. we restrict π to $\pi \in \mathcal{N}$ (otherwise the claim follows from $c \sqsubseteq s$). According to Lemma 1.36 for each position π there is a position β with $s|_\beta = s'|_\pi$ and $c|_\beta = c|_\pi$. Similarly, there is a position β' with $s|_{\beta'} = s'|_{\pi'}$ and $c|_{\beta'} = c|_{\pi'}$.

(a – d) Trivial.

(e) Let $\pi \in \textsc{SPos}(s')$. With Definition 1.10(e) we have $h_c(c|_\pi) = h_c(c|_\beta)$, $h(s|_\beta) = h'(s'|_\pi)$, and $h(s|_\beta) \in \{h_c(c|_\beta), \bot\}$.

(f) Let $\pi \in \textsc{SPos}(s')$. With Definition 1.10(f) we have $h_c(c|_\pi) = h_c(c|_\beta) \subseteq h(s|_\beta) = h'(s'|_\pi)$.

(g) Let $\pi \in \textsc{SPos}(s')$. With Definition 1.10(g) we have $t_c(c|_\pi) = t_c(c|_\beta) \subseteq t(s|_\beta) = t'(s'|_\pi)$.

(h) Let $c|_\pi = \mathsf{null}$. Let $\pi \in \textsc{SPos}(s')$. We have $\mathsf{null} = c|_\pi = c|_\beta$. With Definition 1.10(h) we have $c|_\beta = s|_\beta = s'|_\pi = \mathsf{null}$ or $s|_\beta?$, $s'|_\pi?$, $h(s|_\beta) = f = h'(s'_\pi) \in \textsc{Instances}$, and $\mathrm{dom}(f) = \emptyset$.

(i) Let $\pi \in \textsc{SPos}(s')$. We have $h_c(c|_\pi) = h_c(c|_\beta) = f_c \in \textsc{Instances}$. Furthermore, we have $h(s|_\beta) = f \in \textsc{Instances}$ and $h'(s'_\pi) = f' \in \textsc{Instances}$ with $\mathrm{dom}(f) = \mathrm{dom}(f')$. With Definition 1.10(i) we also have $\mathrm{dom}(f_c) \subseteq \mathrm{dom}(f)$. Thus, the claim follows. Note that the only difference of f and f' may be that $f(v) = r$ whereas $f'(v) = r'$.

(j) Let $\pi \in \textsc{SPos}(s')$. We have $h_c(c|_\pi) = h_c(c|_\beta) = (i_{l,c}, f_c) \in \textsc{Arrays}$. With Definition 1.10(j) we either have $h(s|_\beta) = h'(s'|_\pi) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) = \emptyset$ or $h(s|_\beta) = (i_l, f) \in \textsc{Arrays}$, $h'(s'|_\pi) = (i_l, f') \in \textsc{Arrays}$ with

$\mathrm{dom}(f_c) \supseteq \mathrm{dom}(f) = \mathrm{dom}(f')$. Thus, the claim follows. Note that the only difference of f and f' may be that $f(i) = r$ whereas $f'(i) = r'$.

(k) Let $\pi' \in \mathrm{SP{\scriptsize OS}}(c)$ with $c|_\pi \neq c|_{\pi'}$. Assume $\pi, \pi' \in \mathrm{SP{\scriptsize OS}}(s')$. Let β' with $c|_{\pi'} = c|_{\beta'}$, $s|_{\beta'} = s'|_{\pi'}$. We have $c|_\pi = c|_\beta$ and $s|_\beta = s'|_\pi$. We also have $c|_{\pi'} = c|_{\beta'}$ and $s|_{\beta'} = s'|_{\pi'}$. With Definition 1.10(k) we know $s|_\beta \neq s|_{\beta'}$. Thus, the claim follows.

(l) Let $c|_\pi = c|_{\pi'}$ with $\pi \neq \pi'$ and $h_c(c|_\pi) \in \mathrm{I{\scriptsize NSTANCES}} \cup \mathrm{A{\scriptsize RRAYS}}$. Assume $\pi, \pi' \in \mathrm{SP{\scriptsize OS}}(s')$. Let β' with $c|_{\pi'} = c|_{\beta'}$, $s|_{\beta'} = s'|_{\pi'}$. With Definition 1.10(l) we know $s|_\beta = s|_{\beta'}$ or $s|_\beta =^? s|_{\beta'}$. Thus, with $s|_\beta = s'|_\pi$ and $s|_{\beta'} = s'|_{\pi'}$ the claim follows.

(m) Let $c|_\pi = c|_{\pi'}$ with $\pi \neq \pi'$ and $h_c(c|_\pi) \in \mathrm{I{\scriptsize NSTANCES}} \cup \mathrm{A{\scriptsize RRAYS}}$. Assume $\{\pi, \pi'\} \not\subseteq \mathrm{SP{\scriptsize OS}}(s')$. Assume $s'|_{\overline{\pi}} \neq s'|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. s'. Similar to β, we have $\tilde{\beta}$ with $s|_{\tilde{\beta}} = s'|_{\overline{\pi}}$. Also, let $\tilde{\beta}'$ with $s|_{\tilde{\beta}'} = s'|_{\overline{\pi'}}$. Then we also have $s|_{\tilde{\beta}} \neq s|_{\tilde{\beta}'}$ or $\tilde{\beta}, \tilde{\beta}'$ have different suffixes w.r.t. s. With Definition 1.10(m) and $c \sqsubseteq s$ we then have $s|_{\tilde{\beta}} \curlyvee s|_{\tilde{\beta}'}$. Thus, we also have $s'|_{\overline{\pi}} \curlyvee s'|_{\overline{\pi'}}$ and the claim follows.

(n) Let $\pi = \alpha\tau$ and $\pi' = \alpha\tau'$ with $\tau \neq \varepsilon$ and where τ, τ' have no common intermediate reference from α in c. Assume $c|_\pi = c|_{\pi'}$ where $c|_\pi \in \mathrm{I{\scriptsize NSTANCES}} \cup \mathrm{A{\scriptsize RRAYS}}$.

If $\pi, \pi' \in \mathrm{SP{\scriptsize OS}}(s')$ with Definition 1.10(n) we may have $s|_\beta = s|_{\beta'}$. If so, we also have $s|_\beta = s'|_\pi$ and $s|_{\beta'} = s'|_{\pi'}$, thus the claim follows.

Otherwise, we have $\tilde{\beta}$ with $s|_{\tilde{\beta}} = s'|_{\overline{\alpha}}$. According to Definition 1.10(n) we then have $s|_{\tilde{\beta}} \curlyvee s|_{\tilde{\beta}}$ and, if $\tau' = \varepsilon$, also $s|_{\tilde{\beta}} \circlearrowleft_F$ with $F \subseteq \tau$. Thus, we also have $s'|_{\overline{\alpha}} \curlyvee s'|_{\overline{\alpha}}$ and $s'|_{\overline{\alpha}} \circlearrowleft_F$ and the claim follows.

(o – r) Not applicable, as c is concrete. □

Now, we can finally prove that the equality refinement as presented on page 46 is valid.

Proof. (Equality refinement is valid) Let $r =^? r'$ be the references used in the refinement of state s. Let c be a concrete state with $c \sqsubseteq s$.

- If $\mathrm{refine}(s) = \{s_{\neq}\}$, we need to show $c \sqsubseteq s_{\neq}$.

  As $s_{\neq}$ is identical to s, where we just removed a $=^?$ heap predicate, we just need to check Definition 1.10(l). Thus, assume we have $c|_\pi = c|_{\pi'}$ with $\pi \neq \pi'$ and $h_c(c|_\pi) \in \mathrm{I{\scriptsize NSTANCES}} \cup \mathrm{A{\scriptsize RRAYS}}$. If $\{\pi, \pi'\} \not\subseteq \mathrm{SP{\scriptsize OS}}(s_{\neq})$ the claim follows. Otherwise, if $\pi, \pi' \in \mathrm{SP{\scriptsize OS}}(s_{\neq})$ we need to have $s_{\neq}|_\pi = s_{\neq}|_{\pi'}$ or $s_{\neq}|_\pi =^? s_{\neq}|_{\pi'}$. If this $=^?$ predicate exists in $s_{\neq}$, the claim follows. Otherwise, $\pi \in \Pi, \pi' \in \Pi'$, we removed $r =^? r'$, and we know $\mathrm{intersect}(s[r/r'], \widetilde{s[r'/r]})$ failed. With Lemma 1.33

we know that $c \not\sqsubseteq s[r/r']$ or $c \not\sqsubseteq \widetilde{s[r'/r]}$. With Lemma 1.37 we conclude that c cannot exist, thus the claim is shown for all possible states c.

- If $\text{refine}(s) = \{s_{\neq}, s_{=}\}$ where $s_{=} = \text{intersect}(s[r/r'], \widetilde{s[r'/r]}) \neq ※$, we need to show $c \sqsubseteq s_{\neq}$ or $c \sqsubseteq s_{=}$.

  Let $\Pi = \{\pi \mid s|_\pi = r\}, \Pi' = \{\pi' \mid s|_{\pi'}\}$. If $c|_\pi \neq c|_{\pi'}$ for all $\pi \in \Pi$, $\pi' \in \Pi'$ we have $c \sqsubseteq s_{\neq}$ as above. Otherwise, the claim follows from Lemma 1.37 and Theorem 1.35. □

1.6. Evaluation

Earlier we introduced abstract states (Definition 1.1) that can also be restricted to concrete states (Definition 1.6). An important observation is that states as defined in the *Java Language Specification* can directly be adapted to concrete states as defined in this thesis[2]. Thus, for every computation in a given JAVA BYTECODE program we can create the corresponding concrete states as defined in this thesis. Let $c \xrightarrow{jvm} c'$ denote a single evaluation step on concrete states, i.e., if the current opcode of c is evaluated, the changes are applied to c, resulting in c'.

The main goal is to construct a finite Symbolic Execution Graph for a given program so that every possible computation starting in any state $c \sqsubseteq s$, where s is the start state of the graph, is represented in the graph. Furthermore, we want the Symbolic Execution Graph to represent as few additional computations as possible. Thus, we want to have a finite representation of all possible computations containing very detailed information. A Symbolic Execution Graph with this information can then, for example, be used to show termination of the represented computations.

Even if the start state s was concrete, in order to have a finite representation of arbitrary infinite computations, we need to introduce abstraction. Thus, as soon as a non-concrete abstract state needs to be evaluated, we need to know how to evaluate abstract states. In order to have all possible computations represented in the resulting Symbolic Execution Graph, for $c \xrightarrow{jvm} c'$ with $c \sqsubseteq s$ we need to have $c' \sqsubseteq s'$ where s' is the result of (abstract) evaluation of s (denoted $s \xrightarrow{\text{EVAL}} s'$).

While in [GJS+12] only evaluation of concrete states is specified, we will explain how this specification can be extended to abstract states. This is trivial or simple for some opcodes, but it is not straightforward how evaluation on abstract states should look like in other cases. As a trivial example, consider the NOP opcode. According to the specification the state before evaluating this opcode is identical to the state that results out of evaluation

[2]Here we only consider programs conforming to the limitations mentioned on page 13.

(where just the current opcode is advanced). Thus, when considering abstract states, an adaption is straightforward. Next, we consider the POP opcode which removes the top entry of the operand stack. To compute the desired outcome, we do not need to know the concrete value which is removed from the operand stack. Thus, evaluation of the opcode POP can also easily be adapted to the setting of abstract states.

Slightly more interesting is the IINC opcode which increments an integer variable stored in a given local variable. If the information of the abstract state indicates that the variable contains a literal, we can compute and store the result, also a literal, back into the local variable. However, if we just have the information that the referenced data is an interval of possible values, we need to extend the specification. If we increment by 1 and in a state s a local variable containing a reference i_1 with $h(i_1) = [5, 10]$ needs to be incremented, in the evaluation successor s' with $s \xrightarrow{\text{EVAL}} s'$ we need to consider all possible outcomes of evaluating any of the concrete states represented by s. Thus, since $5 \in [5, 10]$ and incrementing 5 gives 6, in s' we need to have the information that 6 is a possible value for the variable. Similarly, the values 7 to 11 also need to be contained. Thus, a state s' where the variable references $[5, 11]$ would be valid in the sense described above.

However, we also know that the variable can never contain the value 5 after incrementing a value in $[5, 10]$. Thus, a more precise version of s' may have the information where the variable references the interval $[6, 11]$. Note that any of the two variants of s' may be used in the graph, but that the version presented first is less precise. This could lead to additional computations represented by the Symbolic Execution Graph which however do not correspond to concrete computations.

Additionally, we need to consider cases where evaluation is not possible with the information provided in the state. As already mentioned when refinements were introduced, we cannot find a single evaluation successor for a state s containing the information $h(i_1) = [0, 1]$, the current opcode IFEQ (jumping to a certain branch target if the top value on the operand stack is 0), and i_1 on top of the operand stack. Thus, we may need to refine certain parts of the abstract state before we can evaluate.

As a consequence, our goals when evaluating a state s are

(i) to refine s in order to make evaluation possible

(ii) to take care that for $c \sqsubseteq s$, $c \xrightarrow{jvm} c'$, and $s \xrightarrow{\text{EVAL}} s'$ we also have $c' \sqsubseteq s'$

(iii) to take care that the resulting state s' is as precise as possible

Sadly, the specification in [GJS+12] has about 600 pages and does not formalize the semantics of Java Bytecode. Thus, it is out of scope to describe (or formalize) every aspect of the specification in this thesis. For the same reason there also does not exist any formalization of the Java Bytecode semantics that could be used to formally prove correctness of this approach. However, in [KN06] a language related to Java Bytecode, named JINJA, is formalized. The state formalization in [KN06] and the concrete states defined in this thesis are somewhat similar (as both are designed to represent the states possibly created by an actual JVM) and in [BOvEG10] we already presented correctness results based on this formalization. However, in the setting of this thesis, we use a nearly complete subset of JAVA BYTECODE, thus a similar approach is not feasible.

The first aspect mentioned above, regarding refinement, was already explained earlier. Here, especially Fig. 1.25 gives the necessary information. In this thesis we will not provide further details, as using the refinements already defined it is easily possible to refine an abstract state such that abstract interpretation can be performed.

For the second aspect, we would need to re-define the semantics of all opcodes. As already mentioned, we do not provide all details in this thesis. Instead, we argue that for most opcodes, using refinement where necessary, evaluation of abstract states is straightforward. As an example, to evaluate an INVOKEVIRTUAL opcode a null-check needs to be performed, the choice of the invoked method depends on the type of a certain object, and the creation of the new stack frame is rather involved. However, using possibly several refinements the process as outlined in [GJS+12] can easily be implemented for abstract states. This implementation should also deal with the third goal mentioned above, which is trivial for most opcodes and a bit more involved for (integer) arithmetic.

However, in the case of the PUTFIELD opcode which modifies an object instance on the heap, it is not easy to see how the information for other parts of the heap (that may be connected as indicated using heap predicates) need to be adapted. Thus, in the next part of this section, we will introduce the semantics of PUTFIELD on abstract states and give a proof that, indeed, no concrete computation is "lost" when evaluating this opcode on abstract states.

The only other opcode modifying and creating connections on the heap, AASTORE which is used to store a reference into an array, is very similar to PUTFIELD. However, while for object instances we know which field we write into, the array index used in an array write access or the size of the array may be unknown when evaluating the opcode on an abstract state. In Section 1.6.2 the corresponding adaptations are presented.

Similarly, when evaluating the opcode AALOAD, which loads a reference out of an array, it can happen that the connection between the reference read from the array and the array itself cannot be represented explicitly. In order to correctly model these (and other) connections on the heap, we need to add heap predicates. This process is outlined in Section 1.6.3.

The following theorem states that evaluation of abstract states is valid, similar to the definition of valid refinements, in the sense that no computation is lost.

Theorem 1.38 (Evaluation is valid) Let $s, s' \in \textsc{States}$ and $s \xrightarrow{\textsc{Eval}} s'$ be an evaluation. For all concrete states c, c' with $c \sqsubseteq s$ and $c \xrightarrow{jvm} c'$ we have $c' \sqsubseteq s'$.

For the interesting cases of PUTFIELD, AALOAD, and AASTORE this theorem is proven in Theorems 1.41, 1.45, and 1.49. For all other opcodes (and less interesting cases of the three opcodes named above) evaluation is straightforward, as the necessary refinements already dealt with all complications. As we already have shown validity of refinement, there is not much need to also show validity of evaluation for most opcodes.

1.6.1. PUTFIELD

When evaluating the PUTFIELD opcode on abstract states, we must take into account that the changes of the write access can also be observed from variables that share with the object instance modified by the opcode. This is illustrated in Example 1.39.

Example 1.39 In this example, the write access in line 18 also modifies data that can be seen by traversing the list in variable first. As a consequence of the write access, the check in line 22 causes the program to not terminate.

```
1  public class List {
2    Object marker = null;
3    List next;
4
5    public static void main(String[] args) {
6      List list = new List();
7
8      // remember first element of list
9      List first = list;
10
11     // create list of arbitrary length, marker is not set
12     for (int i = 1; i < args.length; i++) {
13       list.next = new List();
14       list = list.next;
15     }
16
17     Object marker = new Object();
18     list.marker = marker; // set marker at end of list
19
20     for (List cur = first; cur != null; cur = cur.next) {
21       // do  not terminate if marker is set
22       while (cur.marker == marker) {}
23     }
24   }
25 }
```

In the case of abstract evaluation, the references for the variables first and list would be connected using a $\searrow\!\!\swarrow$ heap predicate (after executing the loop in lines 12–15 at least twice). Because of this, when evaluating the PUTFIELD opcode corresponding to the write access in line 18, we must consider that a change to the reference of list might also change data reachable from the reference of first. Consequently, we need to add a $\searrow\!\!\swarrow$ heap predicate to model that first might reach marker.

The problem made visible in Example 1.39 is that when evaluating PUTFIELD for an abstract state it does not suffice to only change information related to the two involved references on the operand stack. Instead, also connected references need to be regarded. To properly model the effects of evaluating PUTFIELD we consider four situations. Let r_c, r_p be the two references considered by the PUTFIELD opcode where r_c is the reference being written into some field of r_p. Thus, after the write access, for any $s|_\pi = r_p$ we have $s|_{\pi\, v} = r_c$.

(i) There are successors of r_c, so we need to add joins heap predicates. As one can reach r_c through r_p, it then is also possible to reach the successors of r_c from (abstract) predecessors of r_p.

(ii) There is a cycle/non-tree shape visible from r_c. If there is an abstract predecessor of r_p, this reference then also is an abstract predecessor of the cycle/non-tree shape and, thus, we need to add corresponding heap predicates.

(iii) There is a successor of r_c with a heap predicate allowing a non-tree shape, so we need to add the corresponding heap predicates. This is similar to the previous case. Instead of reaching a realized cycle/non-tree shape and adding the corresponding heap predicates, here we just need to propagate the heap predicates to the abstract predecessors of r_p.

(iv) The write access creates a new cycle/non-tree shape in the heap, so we need to add corresponding heap predicates. There may be a path from an abstract predecessor of r_p to a successor of r_c, so that with the new connection from r_p to r_c a second path is created – thus, we then have a non-tree shape which possibly did not exist before. As a consequence, we might need to add corresponding heap predicates.

The first item corresponds to the case made in Example 1.39. Here, we need to add $r_a \searrow\!\!\swarrow r_b$ for each abstract predecessor r_a of r_p and each successor r_b of r_c. Without this, in abstract states where the connection from the local variable first to the modified list element is only defined using a $\searrow\!\!\swarrow$ heap predicate, the comparison in line 24 of Example 1.39 cannot evaluate to true (and, thus, we could falsely prove termination of the nonterminating algorithm).

The second and third item can be understood best by modifying the previous example, resulting in Example 1.40.

Example 1.40 As in Example 1.39, we remember the beginning of the list. The write access in line 19 causes the list to be cyclic.

```
1  public class List {
2    List next;
3
4    public static void main(String[] args) {
5      List list = new List();
6
7      // remember first element of list
8      List first = list;
9
10     // create list of arbitrary length
11     for (int i = 1; i < args.length; i++) {
12       list.next = new List();
13       list = list.next;
14     }
15
16     // make list cyclic
17     List cycle = new List();
18     cycle.next = cycle;
19     list.next = cycle;
20
21     // iterate over cyclic list
22     for (List cur = first; cur != null; cur = cur.next) {
23     }
24   }
25 }
```

In the case of abstract interpretation, we must also consider the case that first references a cyclic list after the write access in line 19. This is done by adding the corresponding heap predicate to the reference of first. Without this heap predicate the data structure would be known to be acyclic, making it possible to falsely show termination of the loop starting in line 22.

Finally, the fourth item deals with cases where performing the write access causes a previously non-existing non-tree shape or cycle to be created on the heap. Detecting the creation of such shapes is rather involved if parts of the heap are only represented using heap predicates. As an example, in a state with references r_1 to r_4 and $r_1 \searrow\!\!\swarrow r_2$, $r_1 \searrow\!\!\swarrow r_4$, $r_3 \searrow\!\!\swarrow r_4$ evaluating PUTFIELD writing r_3 into a field of r_2 might create a non-tree shape: after the write there may be two paths from r_1 to r_4. The first path may exist due to $r_1 \searrow\!\!\swarrow r_4$. The second path may exist because of $r_1 \searrow\!\!\swarrow r_2$ and $r_3 \searrow\!\!\swarrow r_4$, where the "gap" from r_2 to r_3 is closed by the write access.

We will now formally define which heap predicates are added when PUTFIELD is evaluated. Then we show that the abstract evaluation is correct, i.e., for $c \xrightarrow{jvm} c'$ and $c \sqsubseteq s$ evaluating PUTFIELD with $s \xrightarrow{\text{EVAL}} s'$ also guarantees $c' \sqsubseteq s'$.

In our approach, abstract evaluation of a PUTFIELD opcode writing into a field v of r_p only works if there is no reference r with $r =^? r_p$. This is no real restriction, as we can always use refinement to drop such heap predicates. Similarly, we only consider the cases where r_p references an object instance with a field v. Furthermore, we assume that no exception mentioned in the section *linking exceptions* in the *Java Language Specification* is thrown[3].

Before the actual definition of PUTFIELD evaluation, we need to define two relations $\sim$ and $\rightsquigarrow$ that help identifying related references on the heap.

Definition 1.41 ($\sim_s$) For two references r_a, r_b we have $r_a \sim_s r_b$ iff $r_a =^?_s r_b$ or $r_a \searrow\!\!\swarrow_s r_b$. We write $\sim$ instead of $\sim_s$ if the state s is clear from the context.

Definition 1.42 ($\rightsquigarrow_s$) For two references r_a, r_b we have $r_a \rightsquigarrow_s r_b$ iff there are π, ρ such that $s|_\pi = r_a \wedge (s|_{\pi\rho} \sim_s r_b \vee s|_{\pi\rho} = r_b)$. We write $\rightsquigarrow$ instead of $\rightsquigarrow_s$ if the state s is clear from the context.

In other words, $r_a \sim r_b$ describes that a connection of the references r_a and r_b is allowed by some heap predicate. For $r_a \rightsquigarrow r_b$ we see that one may reach r_b from r_a.

Definition 1.43 (Evaluating PUTFIELD) Let pp_0 be a PUTFIELD opcode writing into a field named v. Let s be the corresponding state where $s = (\langle fr_0, \ldots, fr_n \rangle, h, t, hp, sf, e, ic, \bot)$ and $fr_i = (pp_i, lv_i, os_i)$. Let $s|_{\text{OS}_{0,0}} =: r_c$ and $s|_{\text{OS}_{0,1}} =: r_p$. Let there be no reference r with $r =^? r_p$, let there be no heap predicate $r_p?$ and let $f = h(r_p) \in \textsc{Instances}$ with $v \in \text{dom}(f)$.

Then we define s' with $s \xrightarrow{\text{EVAL}} s'$ as $s' = (\langle fr'_0, fr_1, \ldots, fr_n \rangle, h', t, hp', sf, e, ic, \bot)$ with $fr'_0 = (pp_0, lv_0, os'_0)$. The topmost operand stack os'_0 is identical to os_0, where just the two topmost elements r_c and r_p are removed. For the heap h' we just update

[3]Detecting if one of those exceptions needs to be thrown is rather technical. As a consequence, we decided to not mention the corresponding preconditions in Definition 1.43.

the field information for r_p: $h' = h + \{r_p \mapsto f'\}$ with $f' = f + \{v \mapsto r_c\}$. We define $hp' \supseteq hp$ where we add heap predicates according to the following rules:

(i) for all r_a, r_b with $r_a \curlyvee r_p$ and $r_c \rightsquigarrow r_b$ we add $r_a \curlyvee r_b$

(ii) if there are π, ρ, ρ' with $\rho \neq \rho'$ where $s|_\pi = r_c$ and $s|_{\pi\rho} = s|_{\pi\rho'}$, then we add $r_a \curlyvee r_a$ for all $r_a \curlyvee r_p$

(iii) if there are π, ρ, ρ' with $\rho' \neq \varepsilon$ where $s|_\pi = r_c$ and $s|_{\pi\rho} = s|_{\pi\rho\rho'}$, then we add $r_a \circlearrowleft_F$ for all $r_a \curlyvee r_p$ where F contains all fields in ρ'

(iv) if there are π, ρ with $s|_\pi = r_c$ and $s|_{\pi\rho} \curlyvee s|_{\pi\rho}$, then we add $r_a \curlyvee r_a$ for all $r_a \curlyvee r_p$

(v) if there are π, ρ with $s|_\pi = r_c$ and $s|_{\pi\rho} \circlearrowleft_F$, then we add $r_a \circlearrowleft_F$ for all $r_a \curlyvee r_p$

(vi) if there are r_a, r_b with $r_a \rightsquigarrow r_b$, $r_a \rightsquigarrow r_p$, $r_c \rightsquigarrow r_b$, and the paths from r_a to r_p and r_a to r_b have no common intermediate reference, then we add $r_a \curlyvee r_a$

(vii) if there is r_a with $r_a \rightsquigarrow r_p$, $r_c \rightsquigarrow r_a$ then we add $r_a \circlearrowleft_F$ where F contains v in addition to the fields on the paths from r_a to r_p and r_c to r_a

When adding $o \circlearrowleft_F$ to a state already containing the heap predicate $o \circlearrowleft_{F'}$ this means that after the addition we have $o \circlearrowleft_{F''}$ where $F'' = F \cap F'$.

The heap predicates added as defined in Definition 1.43 can be put into four categories:

(i) abstract predecessors of r_p may reach new parts of the heap

(ii, iii) existing non-tree shapes may be reached from abstract predecessors of r_p

(iv, v) abstract non-tree shapes may be reached from abstract predecessors of r_p

(vi, vii) new non-tree shapes are created and may be reached from predecessors of r_p

Using this definition it is possible to evaluate PUTFIELD opcodes on abstract states. Now we show that such evaluation is correct in the sense that the resulting abstract state also represents concrete states resulting out of a corresponding concrete evaluation of PUTFIELD.

Theorem 1.41 Let s, s' be states as in Definition 1.43 with $s \xrightarrow{\text{EVAL}} s'$. Then for all states c, c' with $c \sqsubseteq s$ and $c \xrightarrow{jvm} c'$ we have $c' \sqsubseteq s'$.

In order to show Theorem 1.41, we first need to discuss the changes caused by evaluating PUTFIELD.

Definition 1.44 ($\boldsymbol{\delta}$) Let $c \xrightarrow{jvm} c'$ be a concrete PUTFIELD evaluation with c, c' as defined in Theorem 1.41. Then let δ denote the function that maps positions in c', which has a shorter operand stack than c, to positions in c. For that, let $|\omega| = 1$.

$$\delta(\omega\pi) = \begin{cases} \mathrm{OS}_{0,j+2}\pi & \text{if } \omega = \mathrm{OS}_{0,j} \\ \omega\pi & \text{otherwise} \end{cases}$$

For any state position π changed by evaluating PUTFIELD (i.e., both references considered by PUTFIELD are part of the path described by π) we can decompose π into three parts. Informally, the first part is the longest prefix leading to the object instance we write into, only visiting unchanged parts of the heap. The third part denotes the suffix of π leading away from the object reference written into the field, also only visiting unchanged parts of the heap. The second part denotes the part in between, i.e., the part of π that goes along the written field at least once. In most cases, the second part only consists of the written field, but in the case of cycles it may be more complicated.

Definition 1.45 (PUTFIELD decomposition) Let $c \xrightarrow{jvm} c'$ be a concrete PUTFIELD evaluation with c, c' as defined in Theorem 1.41. For any $\pi \in \mathrm{SPos}(c')$ with $c'|_\pi \neq c|_{\delta(\pi)}$ we define its PUTFIELD decomposition as $\pi = \tau\beta\eta$ where

- τ is the shortest prefix of π such that both $c'|_\tau = c|_{\mathrm{OS}_{0,1}}$ and $\tau v \trianglelefteq \pi$
- β is the longest position of the form $\beta = v\,\alpha_1\,v\,\alpha_2\,v\,\ldots v\,\alpha_n\,v$ for some $n \geq 0$ where $\tau\beta \trianglelefteq \pi$, $c'|_{\tau v \alpha_j} = c|_{\mathrm{OS}_{0,1}}$, and $c'|_{\tau v \rho} \neq c|_{\mathrm{OS}_{0,1}}$ for all $\varepsilon \neq \rho \triangleleft \alpha_j$ and all $1 \leq j \leq n$. Note that this implies $c'|_{\tau\beta} = c'|_{\tau v} = c|_{\mathrm{OS}_{0,0}}$ and $c'|_\pi = c|_{\mathrm{OS}_{0,0}\eta}$.

We now show that PUTFIELD decompositions can be lifted to abstract states.

Lemma 1.42 (Change of abstract states by PUTFIELD) Let s, s', c, c' as defined in Theorem 1.41. For any $\pi \in \mathrm{SPos}(s') \cap \mathrm{SPos}(c')$ we have:

- if $c'|_\pi = c|_{\delta(\pi)}$, then $s'|_\pi = s|_{\delta(\pi)}$
- if $c'|_\pi \neq c|_{\delta(\pi)}$, then for the corresponding PUTFIELD decomposition $\pi = \tau\beta\eta$ we have $s'|_\tau = s|_{\mathrm{OS}_{0,1}}$, $s'|_{\tau\beta} = s'|_{\tau v} = s|_{\mathrm{OS}_{0,0}}$, and $s'|_\pi = s|_{\mathrm{OS}_{0,0}\eta}$

Proof. According to Definition 1.43 we know $s|_{\text{OS}_{0,1}} \neq \mathsf{null}$ and no heap predicate $s|_{\text{OS}_{0,1}}?$ exists, thus with $c \sqsubseteq s$ we have $c|_{\text{OS}_{0,1}} \neq \mathsf{null}$. Hence, $c'|_\pi = c|_{\delta(\pi)}$ means that the position π is not influenced by the $\mathsf{PUTFIELD}$ instruction. This implies that we also have $s'|_\pi = s|_{\delta(\pi)}$.

Now let $c'|_\pi \neq c|_{\delta(\pi)}$ and let $\pi = \tau\beta\eta$ be the $\mathsf{PUTFIELD}$ decomposition. Since τ is the shortest prefix of π with $c'|_\tau = c|_{\text{OS}_{0,1}}$ and $\tau\, v \trianglelefteq \pi$, this path is not affected by the evaluation, i.e., $c'|_\tau = c|_{\delta(\tau)}$ and $s'|_\tau = s|_{\delta(\tau)}$.

First assume $s'|_\tau \neq s|_{\text{OS}_{0,1}}$. With $c|_{\delta(\tau)} = c'|_\tau = c|_{\text{OS}_{0,1}}$ and $s|_{\delta(\tau)} = s'|_\tau \neq s|_{\text{OS}_{0,1}}$, from $c \sqsubseteq s$ and Definition 1.10(l) we can conclude $s|_{\delta(\tau)} =^? s|_{\text{OS}_{0,1}}$. This contradicts Definition 1.43, where such heap predicates are not allowed. Thus, we have shown that $s'|_\tau = s|_{\text{OS}_{0,1}}$.

Now assume that $s'|_{\tau\beta} \neq s|_{\text{OS}_{0,0}}$. By the definition of the evaluation, we have $s'|_{\tau v} = s|_{\text{OS}_{0,0}}$ (thus, $v \neq \beta$ or we already contradicted the assumption). Recall that $\beta = v\,\alpha_1\, v\,\alpha_2\, v \ldots \alpha_n\, v$, cf. Definition 1.45. From $s'|_{\tau\beta} \neq s|_{\text{OS}_{0,0}}$ we can conclude that there is a j with $s'|_{\tau v \alpha_j} \neq s'|_\tau$. Let j be the minimal such number. As $s'|_{\tau v} = s|_{\text{OS}_{0,0}}$ (and, thus, $s'|_{\tau v\alpha_j} = s|_{\text{OS}_{0,0}\alpha_j}$), we have $s|_{\text{OS}_{0,1}} = s'|_\tau \neq s'|_{\tau v\alpha_j} = s|_{\text{OS}_{0,0}\alpha_j}$. On the other hand, we have $c|_{\text{OS}_{0,1}} = c'|_\tau = c'|_{\tau v\alpha_j} = c|_{\text{OS}_{0,0}\alpha_j}$. Thus, $c \sqsubseteq s$ and Definition 1.10(l) implies $s|_{\text{OS}_{0,1}} =^? s|_{\text{OS}_{0,0}\alpha_j}$. Again, this contradicts Definition 1.43. Thus, we have shown that $s'|_{\tau\beta} = s|_{\text{OS}_{0,0}}$.

As η was not affected by $\mathsf{PUTFIELD}$, $s'|_{\tau\beta} = s|_{\text{OS}_{0,0}}$ implies $s'|_\pi = s|_{\text{OS}_{0,0}\eta}$. □

The following facts are used multiple times, which is why we introduce lemmas for them.

Lemma 1.43 Let s, s', c, c' as defined in Theorem 1.41. Let $\pi \in \text{SP}\textsc{os}(c')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. Let $\pi = \tau\beta\eta$ be the $\mathsf{PUTFIELD}$ decomposition with $\beta = v\,\alpha_1\, v\,\alpha_2\, v \ldots \alpha_n\, v$. If $\tau \in \text{SP}\textsc{os}(s')$ and $\tau\beta \notin \text{SP}\textsc{os}(s')$, then we have $s|_{\text{OS}_{0,1}} \searrow\!\!\swarrow s|_{\text{OS}_{0,0}\widetilde{\alpha_j}}$ for some $\widetilde{\alpha_j} \lhd \alpha_j$.

Proof. There is a minimal j with $\tau\, v\,\alpha_1 \ldots v\,\alpha_j \notin \text{SP}\textsc{os}(s')$. We then also have $\text{OS}_{0,0}\alpha_j \notin \text{SP}\textsc{os}(s)$. As $c|_{\text{OS}_{0,1}} = c|_{\text{OS}_{0,0}\alpha_j}$ by construction of the decomposition and as $c \sqsubseteq s$, with Definition 1.10(m) we have $s|_{\text{OS}_{0,1}} \searrow\!\!\swarrow s|_{\text{OS}_{0,0}\widetilde{\alpha_j}}$ for some $\widetilde{\alpha_j} \lhd \alpha_j$. □

Lemma 1.44 Let s, s', c, c' as defined in Theorem 1.41. Let $\pi \neq \pi' \in \text{SP}\textsc{os}(c')$ with $c'|_\pi = c'|_{\pi'}$, $h_{c'}(c'|_\pi) \in \textsc{Arrays} \cup \textsc{Instances}$, $c'|_\pi \neq c|_{\delta(\pi)}$, $c'|_{\pi'} = c|_{\delta(\pi')}$, and $\pi \notin$

$\text{SPos}(s')$. Let $\pi = \tau\beta\eta$ be the PUTFIELD decomposition with $\beta = v\,\alpha_1\,v\,\alpha_2\,v\ldots\alpha_n\,v$. Then we have $s|_{\text{OS}0,0} \rightsquigarrow s|_{\overline{\delta(\pi')}}$.

Proof. If $\delta(\pi'), \text{OS}_{0,0}\eta \in \text{SPos}(s)$, by $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\delta(\pi')} = s|_{\text{OS}_{0,0}\eta}$ or $s|_{\delta(\pi')} =^? s|_{\text{OS}_{0,0}\eta}$. If $\{\delta(\pi'), \text{OS}_{0,0}\eta\} \not\subseteq \text{SPos}(s)$, by $c \sqsubseteq s$ and Definition 1.10(l) we may have $s|_{\overline{\delta(\pi')}} \bigvee s|_{\text{OS}_{0,0}\widetilde{\eta}}$ for some $\widetilde{\eta} \lhd \eta$. If this heap predicate does not exist, we have $s|_{\overline{\delta(\pi')}} = s|_{\overline{\text{OS}_{0,0}\eta}}$. In all cases we have $s|_{\text{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. □

Now, we can finally prove that evaluation of PUTFIELD as stated on page 77 is correct.

Proof. (PUTFIELD is correct) Let c, c', s, s' be states as defined in Theorem 1.41. We show the claim by showing the individual items of Definition 1.10. Let $\pi, \pi' \in \text{SPos}(c')$.

In order to be able to focus on the important aspects of the proof, namely the cases where a position was changed by the PUTFIELD operation, we first show correctness for the cases where all positions are left unchanged.

For that, assume $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. Then we also have $s'|_\pi = s|_{\delta(\pi)}$ and $s'|_{\pi'} = s|_{\pi'}$ by Lemma 1.42.

(a – c) Trivial.

(d) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(d) we also have $c'|_\pi = c|_{\delta(\pi)} = s|_{\delta(\pi)} = s'|_\pi$. Otherwise, $c'|_\pi \neq c|_{\delta(\pi)}$.

(e) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(e) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\delta(\pi)}) = h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi)$ or $h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi) = \bot$.

(f) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(f) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\delta(\pi)}) \subseteq h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi)$.

(g) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(g) we also have $t_{c'}(c'|_\pi) = t_c(c|_{\delta(\pi)}) \subseteq t(s|_{\delta(\pi)}) = t_{s'}(s'|_\pi)$.

(h) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(h) we also have $c'|_\pi = c|_{\delta(\pi)} = s|_{\delta(\pi)} = s'|_\pi = \mathsf{null}$ or $h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi) = f$ with $\text{dom}(f) = \emptyset$.

(i) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(i) we also have $h_{c'}(c'|_\pi) = f_{c'}, h_c(c|_{\delta(\pi)}) = f_c$ with $\text{dom}(f_{c'}) = \text{dom}(f_c)$, $h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi) = f$ with $\text{dom}(f_{c'}) \supseteq \text{dom}(f)$.

(j) Assume $\pi \in \text{SPos}(s')$. With $c \sqsubseteq s$ and Definition 1.10(j) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\delta(\pi)}) = (i'_l, f')$ and

- $h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi) = (i_l, f)$ with $\mathrm{dom}(f') \supseteq \mathrm{dom}(f)$, or
- $h(s|_{\delta(\pi)}) = h_{s'}(s'|_\pi) = f$ with $\mathrm{dom}(f) = \emptyset$.

(k) Assume $\pi, \pi' \in \mathrm{SP{\small OS}}(s')$. Let $c'|_\pi \neq c'|_{\pi'}$. With $c \sqsubseteq s$ and Definition 1.10(k) we also have $s|_{\delta(\pi)} \neq s|_{\delta(\pi')}$, thus $s'|_\pi \neq s'|_{\pi'}$.

(l) Let $\pi \neq \pi'$, $c'|_\pi = c'|_{\pi'}$ with $h_{c'}(c'|_\pi) \in \mathrm{I{\small NSTANCES}} \cup \mathrm{A{\small RRAYS}}$. Assume $\pi, \pi' \in \mathrm{SP{\small OS}}(s')$. With $c \sqsubseteq s$ and Definition 1.10(l) we also have $s|_{\delta(\pi)} = s|_{\delta(\pi')}$ or $s|_{\delta(\pi)} =^? s|_{\delta(\pi')}$, thus $s'|_\pi = s'|_{\pi'}$ or $s'|_\pi =^? s'|_{\pi'}$.

(m) Let $\pi \neq \pi'$, $c'|_\pi = c'|_{\pi'}$ with $h_{c'}(c'|_\pi) \in \mathrm{I{\small NSTANCES}} \cup \mathrm{A{\small RRAYS}}$. Assume $\{\pi, \pi'\} \not\subseteq \mathrm{SP{\small OS}}(s')$. If $s|_{\overline{\delta(\pi)}} \neq s|_{\overline{\delta(\pi')}}$ or $\delta(\pi), \delta(\pi')$ have different suffixes w.r.t. s, then with $c \sqsubseteq s$ and Definition 1.10(m) we also have $s|_{\overline{\delta(\pi)}} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$. Thus, if $s'|_{\overline{\pi}} \neq s'|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. s', we also have $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

(n) Let $\pi = \alpha\rho$ and $\pi' = \alpha\rho'$ with $\rho \neq \varepsilon$, where ρ, ρ' have no common intermediate reference from α in c'. Let $c'|_\pi = c'|_{\pi'}$.

- Assume $\pi, \pi' \in \mathrm{SP{\small OS}}(s')$. With $c \sqsubseteq s$ and Definition 1.10(n) we also have $s|_{\delta(\pi)} = s|_{\delta(\pi')}$, thus $s'|_\pi = s'|_{\pi\rho}$.
- Assume $\{\pi, \pi'\} \not\subseteq \mathrm{SP{\small OS}}(s')$. With $c \sqsubseteq s$ and Definition 1.10(n) we also have $s|_{\overline{\delta(\pi)}} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$ and, if $\rho' = \varepsilon$, also $s|_{\overline{\delta(\pi)}} \circlearrowleft_F$ with $F \subseteq \rho$. Thus, we also have $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$ and, if necessary, $s'|_{\overline{\pi}} \circlearrowleft_F$.

(o – r) Not applicable, as c' is concrete

Now we consider the remaining cases, where the positions are changed by the PUTFIELD operation. According to Lemma 1.42 for $c'|_\pi \neq c|_{\delta(\pi)}$ there is a position η such that $c'|_\pi = c|_{\mathrm{OS}_{0,0}\eta}$ and $s'|_\pi = s|_{\mathrm{OS}_{0,0}\eta}$. Similarly, for $c'|_{\pi'} \neq c|_{\delta(\pi')}$ there is a position η' such that $c'|_{\pi'} = c|_{\mathrm{OS}_{0,0}\eta'}$ and $s'|_{\pi'} = s|_{\mathrm{OS}_{0,0}\eta'}$.

(a – c) Trivial.

(d) Assume $\pi \in \mathrm{SP{\small OS}}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(d) we also have $c'|_\pi = c|_{\mathrm{OS}_{0,0}\eta} = s|_{\mathrm{OS}_{0,0}\eta} = s'|_\pi$.

(e) Assume $\pi \in \mathrm{SP{\small OS}}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(e) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\mathrm{OS}_{0,0}\eta}) = h(s|_{\mathrm{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi)$ or $h(s|_{\mathrm{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi) = \perp$.

(f) Assume $\pi \in \mathrm{SP{\small OS}}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(f) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\mathrm{OS}_{0,0}\eta}) \subseteq h(s|_{\mathrm{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi)$.

(g) Assume $\pi \in \mathrm{SP{\small OS}}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(g) we also have $t_{c'}(c'|_\pi) = t_c(c|_{\mathrm{OS}_{0,0}\eta}) \subseteq t(s|_{\mathrm{OS}_{0,0}\eta}) = t_{s'}(s'|_\pi)$.

(h) Assume $\pi \in \textsc{SPos}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(h) we also have $s|_{\text{OS}_{0,0}\eta} = s'|_\pi = \mathsf{null}$ or $h(s|_{\text{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi) = f \in \textsc{Instances}$ with $\text{dom}(f) = \emptyset$.

(i) Assume $\pi \in \textsc{SPos}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(i) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\text{OS}_{0,0}\eta}) = f'$, $h(s|_{\text{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi) = f$ with $\text{dom}(f') \supseteq \text{dom}(f)$.

(j) Assume $\pi \in \textsc{SPos}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$. With $c \sqsubseteq s$ and Definition 1.10(j) we also have $h_{c'}(c'|_\pi) = h_c(c|_{\text{OS}_{0,0}\eta}) = (i'_l, f')$ and

- $h(s|_{\text{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi) = (i_l, f)$ with $\text{dom}(f') \supseteq \text{dom}(f)$, or
- $h(s|_{\text{OS}_{0,0}\eta}) = h_{s'}(s'|_\pi) = f$ with $\text{dom}(f) = \emptyset$.

(k) Assume $\pi, \pi' \in \textsc{SPos}(s')$. Let $c'|_\pi \neq c'|_{\pi'}$.

- Assume $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. We also have $s'|_\pi = s|_{\delta(\pi)}$ With $c \sqsubseteq s$ and Definition 1.10(k) we also have $s|_{\delta(\pi)} \neq s|_{\text{OS}_{0,0}\eta'}$, thus $s'|_\pi \neq s'|_{\pi'}$.
- Assume $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. This case is analogous to the previous case.
- Assume $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. Then with $c \sqsubseteq s$ and Definition 1.10(k) we also have $s|_{\text{OS}_{0,0}\eta} \neq s|_{\text{OS}_{0,0}\eta'}$, thus $s'|_\pi \neq s'|_{\pi'}$.

(l) Let $\pi \neq \pi'$, $c'|_\pi = c'|_{\pi'}$ with $h_{c'}(c'|_\pi) \in \textsc{Instances} \cup \textsc{Arrays}$. Assume $\pi, \pi' \in \textsc{SPos}(s')$.

- Assume $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. We also have $s'|_\pi = s|_{\delta(\pi)}$. Then with $c \sqsubseteq s$ and Definition 1.10(l) we also have $s|_{\delta(\pi)} = s|_{\text{OS}_{0,0}\eta'}$ or $s|_{\delta(\pi)} =^? s|_{\text{OS}_{0,0}\eta'}$, thus $s'|_\pi = s'|_{\pi'}$ or $s'|_\pi =^? s'|_{\pi'}$.
- Assume $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. This case is analogous to the previous case.
- Assume $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. With $c \sqsubseteq s$ and Definition 1.10(l) we also have $s|_{\text{OS}_{0,0}\eta} = s|_{\text{OS}_{0,0}\eta'}$ or $s|_{\text{OS}_{0,0}\eta} =^? s|_{\text{OS}_{0,0}\eta'}$, thus $s'|_\pi = s'|_{\pi'}$ or $s'|_\pi =^? s'|_{\pi'}$.

(m) Let $\pi \neq \pi'$, $c'|_\pi = c'|_{\pi'}$ with $h_{c'}(c'|_\pi) \in \textsc{Instances} \cup \textsc{Arrays}$. W.l.o.g. assume $\pi \notin \textsc{SPos}(s')$. We handle both the cases that $\pi' \in \textsc{SPos}(s')$ and $\pi' \notin \textsc{SPos}(s')$ here.

- Assume we have $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. Let $\pi = \tau\beta\eta$ be the PUTFIELD decomposition with $\beta = v\,\alpha_1\,v\,\alpha_2\,v \ldots \alpha_n\,v$.

- We have $\tau\beta \in \mathrm{SPOS}(s')$ and $\pi = \tau\beta\eta \notin \mathrm{SPOS}(s')$. Then also $\mathrm{OS}_{0,0}\eta \notin \mathrm{SPOS}(s)$, because otherwise the object at position $h(s|_{\mathrm{OS}_{0,0}\eta})$ would have been written to position $\pi = \tau\beta\eta$ in s'. We have $c|_{\delta(\pi')} = c'|_{\pi'} = c'|_{\pi} = c|_{\mathrm{OS}_{0,0}\eta}$ and as $c \sqsubseteq s$, with Definition 1.10(m) we may have $s|_{\overline{\delta(\pi')}} \searrow\!\!\swarrow s|_{\overline{\mathrm{OS}_{0,0}\eta}}$. Note that $\overline{\mathrm{OS}_{0,0}\eta}_s = \mathrm{OS}_{0,0}\widetilde{\eta}$ for some $\widetilde{\eta} \trianglelefteq \eta$. Thus we may also have $s'|_{\overline{\pi'}} \searrow\!\!\swarrow s'|_{\tau\beta\widetilde{\eta}}$ and hence, $s'|_{\overline{\pi'}} \searrow\!\!\swarrow s'|_{\overline{\pi}}$. Otherwise, if the joins heap predicate does not exist, we have $s|_{\overline{\delta(\pi')}} = s|_{\overline{\mathrm{OS}_{0,0}\eta}}$ where $\delta(\pi'), \mathrm{OS}_{0,0}\eta$ have the same suffix w.r.t. s. Thus, we also have $s'|_{\overline{\pi'}} = s'|_{\overline{\tau\beta\eta}}$ where $\pi', \tau\beta\eta$ have the same suffix w.r.t. s'.
- We have $\tau \in \mathrm{SPOS}(s')$ and $\tau\beta \notin \mathrm{SPOS}(s')$. With Lemma 1.43 we have $s|_{\mathrm{OS}_{0,1}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,0}\widetilde{\alpha_j}}$ for some $1 \leq j \leq n$ and $\widetilde{\alpha_j} \trianglelefteq \alpha_j$. With Lemma 1.44 we also have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. Thus, Definition 1.43(i) requires $s'|_{\tau v \widetilde{\alpha_j}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$. Hence, $s'|_{\tau\, v\, \alpha_1 \ldots v\, \widetilde{\alpha_j}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$ and thus, $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.
- We have $\tau \notin \mathrm{SPOS}(s')$. Then also $\delta(\tau) \notin \mathrm{SPOS}(s)$ and as $c|_{\delta(\tau)} = c|_{\mathrm{OS}_{0,1}}$ and $c \sqsubseteq s$, with Definition 1.10(m) we have $s|_{\overline{\delta(\tau)}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,1}}$. With Lemma 1.44 we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. Thus, Definition 1.43(i) requires $s'|_{\overline{\tau}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

- Assume we have $c'|_{\pi} = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. This case is analogous to the previous case.
- Assume we have $c'|_{\pi} \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. Let $\pi = \tau\beta\eta$ and $\pi' = \tau'\beta'\eta'$ be the $\mathsf{PUTFIELD}$ decomposition with $\beta = v\,\alpha_1\,v\,\alpha_2\,v \ldots \alpha_n\,v$ and $\beta' = v\,\alpha'_1\,v\,\alpha'_2\,v \ldots \alpha'_{n'}\,v$.
  - We have $\tau\beta \in \mathrm{SPOS}(s')$ and $\tau'\beta' \in \mathrm{SPOS}(s')$. Then $\mathrm{OS}_{0,0}\eta \notin \mathrm{SPOS}(s)$ and as $c|_{\mathrm{OS}_{0,0}\eta} = c|_{\mathrm{OS}_{0,0}\eta'}$ and $c \sqsubseteq s$ with Definition 1.10(m) we may have $s|_{\mathrm{OS}_{0,0}\widetilde{\eta}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,0}\widetilde{\eta}'}$. Thus, we have $s'|_{\tau\beta\widetilde{\eta}} \searrow\!\!\swarrow s'|_{\tau'\beta'\widetilde{\eta}'}$ for some $\widetilde{\eta} \trianglelefteq \eta$ and $\widetilde{\eta}' \trianglelefteq \eta'$ and hence, $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

    If this joins heap predicate does not exist, we know $s|_{\overline{\mathrm{OS}_{0,0}\eta}} = s|_{\overline{\mathrm{OS}_{0,0}\eta'}}$ where $\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta'$ have the same suffix w.r.t. s. Thus, we also know $s'|_{\overline{\pi}} = s'|_{\overline{\pi'}}$ where π, π' have the same suffix w.r.t. s'. Hence, the missing joins heap predicate is not needed.
  - We have $\tau\beta \notin \mathrm{SPOS}(s')$, $\tau \in \mathrm{SPOS}(s')$, and $\tau'\beta' \in \mathrm{SPOS}(s')$. Then with Lemma 1.43 we have $s|_{\mathrm{OS}_{0,1}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,0}\widetilde{\alpha_j}}$ for some $1 \leq j \leq n$ and $\widetilde{\alpha_j} \trianglelefteq \alpha_j$. So Definition 1.43(i) requires $s'|_{\tau v \widetilde{\alpha_j}} \searrow\!\!\swarrow s'|_{\tau'\beta'\widetilde{\eta}'}$. Hence, $s'|_{\tau\, v\, \alpha_1 \ldots v\, \widetilde{\alpha_j}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$ and thus, $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.
  - We have $\tau\beta \in \mathrm{SPOS}(s')$, $\tau'\beta' \notin \mathrm{SPOS}(s')$, and $\tau' \in \mathrm{SPOS}(s')$. This case is analogous to the previous case.

- We have $\tau\beta \notin \textsc{SPos}(s')$, $\tau \in \textsc{SPos}(s')$, $\tau'\beta' \notin \textsc{SPos}(s')$, and $\tau' \in \textsc{SPos}(s')$. Then with Lemma 1.43 we have $s|_{\mathrm{OS}_{0,1}} \Vvdash s|_{\mathrm{OS}_{0,0}\widetilde{\alpha_j}}$ for some $1 \leq j \leq n$ and $\widetilde{\alpha_j} \trianglelefteq \alpha_j$. Furthermore, there is a minimal j' with $\tau'\, v\, \alpha'_1 \dots v\, \alpha'_{j'} \notin \textsc{SPos}(s')$. Thus we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\mathrm{OS}_{0,0}\widetilde{\alpha'_{j'}}}$ for some $\widetilde{\alpha'_{j'}} \trianglelefteq \alpha'_{j'}$.

  Thus, Definition 1.43(i) requires $s'|_{\tau v \widetilde{\alpha_j}} \Vvdash s'|_{\tau' v \widetilde{\alpha'_{j'}}}$. Hence, $s'|_{\tau\, v\, \alpha_1 \dots v', \widetilde{\alpha_j}} \Vvdash s'|_{\tau' v \alpha'_1 \dots v \alpha'_{j'}}$ and thus, $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$.

- We have $\tau \notin \textsc{SPos}(s')$ and $\tau'\beta' \in \textsc{SPos}(s')$. As $c|_{\delta(\tau)} = c|_{\mathrm{OS}_{0,1}}$ and $c \sqsubseteq s$ with Definition 1.10(m) we have $s|_{\overline{\delta(\tau)}} \Vvdash s|_{\mathrm{OS}_{0,1}}$. We also get $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\mathrm{OS}_{0,0}\widetilde{\eta}'}$ for some $\widetilde{\eta}' \trianglelefteq \eta$. Thus, Definition 1.43(i) requires $s'|_{\overline{\tau}} \Vvdash s'|_{\tau'\beta'\widetilde{\eta}'}$ and thus, $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$.

- We have $\tau\beta \in \textsc{SPos}(s')$ and $\tau' \notin \textsc{SPos}(s')$. This case is analogous to the previous case.

- We have $\tau \notin \textsc{SPos}(s')$, $\tau'\beta' \notin \textsc{SPos}(s')$, and $\tau' \in \textsc{SPos}(s')$. As $c|_{\delta(\tau)} = c|_{\mathrm{OS}_{0,1}}$ and $c \sqsubseteq s$, with Definition 1.10(m) we have $s|_{\overline{\delta(\tau)}} \Vvdash s|_{\mathrm{OS}_{0,1}}$.

  Furthermore, for there is a minimal j' with $\tau'\, v\, \alpha'_1 \dots v\, \alpha'_{j'} \notin \textsc{SPos}(s')$. Thus we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\mathrm{OS}_{0,0}\widetilde{\alpha'_{j'}}}$ for some $\widetilde{\alpha'_{j'}} \triangleleft \alpha'_{j'}$. Due to Definition 1.43(i), we have $s'|_{\overline{\tau}} \Vvdash s'|_{\tau' v \widetilde{\alpha'_{j'}}}$. Hence, $s'|_{\overline{\tau}} \Vvdash s'|_{\tau' v\, \alpha'_1 \dots v\, \widetilde{\alpha'_{j'}}}$ and thus, $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$.

- We have $\tau\beta \notin \textsc{SPos}(s')$, $\tau \in \textsc{SPos}(s')$, and $\tau' \notin \textsc{SPos}(s')$. This case is analogous to the previous case.

- We have $\tau \notin \textsc{SPos}(s')$ and $\tau' \notin \textsc{SPos}(s')$. Thus, with $c \sqsubseteq s$ and Definition 1.10(m) we may have $s|_{\overline{\delta(\tau)}} \Vvdash s|_{\overline{\delta(\tau')}}$ and hence, $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$. If this joins heap predicate does not exist, we know $s|_{\overline{\delta(\tau)}} = s|_{\overline{\delta(\tau')}}$ where $\delta(\tau), \delta(\tau')$ have the same suffix w.r.t. s, thus we also have $s'|_{\overline{\tau}} = s'|_{\overline{\tau'}}$ where τ, τ' have the same suffix w.r.t. s'. If $\beta = \beta'$ and $\eta = \eta'$, we do not need to have $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$.

  Otherwise, with $c \sqsubseteq s$ and $s|_{\delta(\tau)} = s|_{\mathrm{OS}_{0,1}}$ we know $s|_{\overline{\delta(\tau)}} \Vvdash s|_{\mathrm{OS}_{0,1}}$. If $\beta \neq \beta'$ we know that there is $\rho \neq \varepsilon$ with $v\, \rho = \beta$ or $v\, \rho = \beta'$. Thus, as $c|_{\mathrm{OS}_{0,0}\rho\, v} = c|_{os_{0,0}}$, with $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\mathrm{OS}_{0,0}} = s|_{\mathrm{OS}_{0,0}\rho\, v}$ or $s|_{\mathrm{OS}_{0,0}} \Vvdash s|_{\mathrm{OS}_{0,0}}$. With Definition 1.43(ii,iv), we also add $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$.

  If $\beta = \beta'$ and $\eta \neq \eta'$, with $c|_{\mathrm{OS}_{0,0}\eta} = c|_{\mathrm{OS}_{0,0}\eta'}$, $c \sqsubseteq s$, and Definition 1.10(n) we have $s|_{\mathrm{OS}_{0,0}\eta} = s|_{\mathrm{OS}_{0,0}\eta'}$ or $s|_{\mathrm{OS}_{0,0}\alpha} \Vvdash s|_{\mathrm{OS}_{0,0}\alpha}$ for some α. Thus, with Definition 1.43(ii,iv) we add $s'|_{\overline{\pi}} \Vvdash s'|_{\overline{\pi'}}$.

(n) Let $\pi = \alpha\rho, \pi' = \alpha\rho'$ where ρ, ρ' have no common intermediate reference from α in c' and let $\rho \neq \varepsilon$. Let $c'|_\pi = c'|_{\pi'}$.

- Assume $\pi, \pi' \in \textsc{SPos}(s')$.
  - If $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$ we also have $s'|_\pi = s|_{\delta(\pi)}$, $c'|_{\pi'} = c|_{\mathrm{OS}_{0,0}\eta'}$, and $s'|_{\pi'} = s|_{\mathrm{OS}_{0,0}\eta'}$ by Lemma 1.42. As $c|_{\mathrm{OS}_{0,0}\eta'} = c|_{\delta(\pi)}$ and $c \sqsubseteq s$ with Definition 1.10(n) we may have $s|_{\mathrm{OS}_{0,0}\eta'} = s|_{\delta(\pi)}$, thus we also have $s'|_{\pi'} = s'|_\pi$.

    Otherwise, we have $s|_{\mathrm{OS}_{0,0}\eta'} =^? s|_{\delta(\pi)}$. With $c'|_{\alpha\rho} = c'|_\pi = c|_{\delta(\pi)} = c|_{\delta(\alpha\rho)}$ we also have $c'|_\alpha = c|_{\delta(\alpha)}$ and $s'|_\alpha = s|_{\delta(\alpha)}$. Furthermore, we have $c|_{\delta(\alpha)} \rightsquigarrow c|_{\mathrm{OS}_{0,1}}$, as $\pi' = \alpha\rho'$ and $c|_{\delta(\alpha\rho')} \neq c'|_{\alpha\rho'}$. With $c \sqsubseteq s$ we then also have $s|_{\delta(\alpha)} \rightsquigarrow s|_{\mathrm{OS}_{0,1}}$.

    With $s|_{\delta(\alpha)} \rightsquigarrow s|_{\mathrm{OS}_{0,1}}$, $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\delta(\pi)}$ (as $s|_{\mathrm{OS}_{0,0}\eta'} =^? s|_{\delta(\pi)}$) and $s|_{\delta(\alpha)} \rightsquigarrow s|_{\delta(\alpha\rho)} = s|_{\delta(\pi)}$ Definition 1.43(vi) requires $s'|_\alpha \bigvee s'|_\alpha$.

    We cannot have $\rho' = \varepsilon$, as $c'|_{\pi'} \neq c'|_{\delta(\pi')}$ must hold.
  - If $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$ we have a situation very similar to the one in the previous case. However, here we may have $\rho' = \varepsilon$, thus $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\delta(\pi')} = s|_{\delta(\alpha)}$. In this case, according to Definition 1.43(vii), we added $s'|_\alpha \circlearrowleft_F$ with $F \subseteq \rho$ (we only added the written field, the fields on the path from $s|_{\delta(\alpha)}$ to $s|_{\mathrm{OS}_{0,1}}$, and the fields on the path from $s|_{\mathrm{OS}_{0,0}}$ to $s|_{\delta(\pi)}$ – all these are contained in ρ).
  - If $c'|_\pi \neq c|_{\delta(\pi)}$, $c'|_{\pi'} \neq c|_{\delta(\pi')}$ we also have $c'|_\pi = c|_{\mathrm{OS}_{0,0}\eta}$, $s'|_\pi = s|_{\mathrm{OS}_{0,0}\eta}$, $c'|_{\pi'} = c|_{\mathrm{OS}_{0,0}\eta'}$, and $s'|_{\pi'} = s|_{\mathrm{OS}_{0,0}\eta'}$ by Lemma 1.42. As $c|_{\mathrm{OS}_{0,0}\eta} = c|_{\mathrm{OS}_{0,0}\eta'}$ and $c \sqsubseteq s$ we may have $s|_{\mathrm{OS}_{0,0}\eta} = s|_{\mathrm{OS}_{0,0}\eta'}$, thus we also have $s'|_\pi = s'|_{\pi'}$.

    Otherwise, we have $s'|_\pi =^? s'|_{\pi'}$. If $c'|_\alpha = c|_{\delta(\alpha)}$ we also have $s'|_\alpha = s|_{\delta(\alpha)}$. As $\pi = \alpha\rho$ and $\pi' = \alpha\rho'$, we know that $c|_{\mathrm{OS}_{0,1}}$ is a common intermediate reference of ρ, ρ' for α in c'. Thus, we do not need to have $s'|_{\overline{\pi}} \bigvee s'|_{\overline{\pi'}}$.

    If $c'|_\alpha \neq c|_{\delta(\alpha)}$, let $\alpha = \tau_\alpha\beta_\alpha\eta_\alpha$ be the $\mathsf{PUTFIELD}$ decomposition of α. Then we also have $c'|_\alpha = c|_{\mathrm{OS}_{0,0}\eta_\alpha}$, and $s'|_\alpha = s|_{\mathrm{OS}_{0,0}\eta_\alpha}$. With $c \sqsubseteq s$ and Definition 1.10(n) have $s|_{\mathrm{OS}_{0,0}\eta_\alpha} \bigvee s|_{\mathrm{OS}_{0,0}\eta_\alpha}$ and, if $\rho' = \varepsilon$, $s|_{\mathrm{OS}_{0,0}\eta_\alpha} \circlearrowleft_F$ with $F \subseteq \rho$. Thus, we also have $s'|_\alpha \bigvee s'|_\alpha$ and, if $\rho' = \varepsilon$, also $s'|_\alpha \circlearrowleft_F$ with $F \subseteq \rho$.
- Otherwise, we have $\{\pi, \pi'\} \not\subseteq \textsc{SPos}(s')$. W.l.o.g. assume $\pi \notin \textsc{SPos}(s')$. We handle both the cases that $\pi' \in \textsc{SPos}(s')$ and $\pi \notin \textsc{SPos}(s')$ here. In all cases we need to show $s'|_{\overline{\alpha}} \bigvee s'|_{\overline{\alpha}}$ and, if $\rho' = \varepsilon$, also $s'|_{\overline{\alpha}} \circlearrowleft_F$ with $F \subseteq \rho$.

If $c'|_\pi \neq c|_{\delta(\pi)}$, let $\pi = \tau\beta\eta$ be the PUTFIELD decomposition with $\beta = v\,\alpha_1\,v\,\alpha_2\,v\ldots\alpha_n\,v$ Similarly, if $c'|_{\pi'} \neq c|_{\delta(\pi')}$ let $\pi' = \tau'\beta'\eta'$ and $\beta' = v\,\alpha'_1\,v\,\alpha'_2\,v\ldots\alpha'_{n'}\,v$.

– Assume we have $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. Hence, with $c \sqsubseteq s$, we have $s|_{\overline{\delta(\alpha)}} \leadsto s|_{\mathrm{OS}_{0,1}}$. Similarly, we have $s|_{\overline{\delta(\alpha)}} \leadsto s|_{\overline{\delta(\pi')}} = s|_{\overline{\delta(\alpha\rho')}}$. With Lemma 1.44 we have $s|_{\mathrm{OS}_{0,0}} \leadsto s|_{\overline{\delta(\pi')}}$. Thus, Definition 1.43(vi) requires $s'|_{\overline{\alpha}} \searrow\!\!\swarrow s'|_{\overline{\alpha}}$.

  If $\rho' = \varepsilon$, we have $s|_{\overline{\delta(\alpha)}} = s|_{\overline{\delta(\pi')}}$. Thus, we have $s|_{\mathrm{OS}_{0,0}} \leadsto s|_{\overline{\delta(\alpha)}}$. According to Definition 1.43(vii) we then have $s'|_{\overline{\alpha}}\circlearrowleft_F$ with $F \subseteq \rho$ (we only added the written field, the fields on the path from $s|_{\overline{\delta(\alpha)}}$ to $s|_{\mathrm{OS}_{0,1}}$, and the fields on the path from $s|_{\mathrm{OS}_{0,0}}$ to $s|_{\overline{\delta(\pi')}}$ – all these are contained in ρ).

– Assume we have $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. Hence, with $c \sqsubseteq s$, we have $s|_{\overline{\delta(\alpha)}} \leadsto s|_{\mathrm{OS}_{0,1}}$. Similarly, we have $s|_{\overline{\delta(\alpha)}} \leadsto s|_{\overline{\delta(\pi)}} = s|_{\overline{\delta(\alpha\rho)}}$. With Lemma 1.44 we have $s|_{\mathrm{OS}_{0,0}} \leadsto s|_{\overline{\delta(\pi)}}$. Thus, Definition 1.43(vi) requires $s'|_{\overline{\alpha}} \searrow\!\!\swarrow s'|_{\overline{\alpha}}$.

  If $\rho' = \varepsilon$, we have $c|_{\delta(\alpha)} = c|_{\delta(\pi)}$. Thus, with $c \sqsubseteq s$, Definition 1.10(n), $s|_{\overline{\delta(\alpha)}} = s'|_{\overline{\alpha}}$, and $s|_{\overline{\delta(\pi)}} = s'|_{\overline{\pi}}$, we do not need to add a heap predicate.

– Assume we have $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. As ρ, ρ' have no common intermediate reference, and both π, π' are affected by the PUTFIELD operation, we know $c|_{\delta(\alpha)} \neq c'|_\alpha$ and $\tau = \tau' \lhd \alpha$. Thus, let $\alpha = \tau\beta_\alpha\eta_\alpha$ be the PUTFIELD decomposition of α.

  * We have $\tau\beta \in \mathrm{SPos}(s')$ and $\tau'\beta' \in \mathrm{SPos}(s')$. We have $c|_{\mathrm{OS}_{0,0}\eta} = c|_{\mathrm{OS}_{0,0}\eta'}$ and $\{\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta'\} \not\subseteq \mathrm{SPos}(s)$. With $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}} \searrow\!\!\swarrow s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}}$. Thus, we also have $s'|_{\overline{\alpha}} \searrow\!\!\swarrow s'|_{\overline{\alpha}}$. If $\rho' = \varepsilon$, we also have $s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}}\circlearrowleft_F$ and $s'|_{\overline{\alpha}}\circlearrowleft_F$ where $F \subseteq \rho$.

  * We have $\tau\beta \notin \mathrm{SPos}(s')$, $\tau \in \mathrm{SPos}(s')$, and $\tau'\beta' \in \mathrm{SPos}(s')$. We also have $\alpha \lhd \tau\beta$, as $\tau'\beta' \in \mathrm{SPos}(s')$. Thus, $\tau \lhd \alpha \lhd \tau\beta$. From this we conclude, with $c|_{\mathrm{OS}_{0,1}} = c|_{\mathrm{OS}_{0,0}\alpha_1\,v\ldots\alpha_n}$ and $c \sqsubseteq s$, that $s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}} \leadsto s|_{\mathrm{OS}_{0,1}}$. We also have $s|_{\mathrm{OS}_{0,0}} \leadsto s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}}$. Thus, with Definition 1.43(vi) we have $s'|_{\overline{\alpha}} \searrow\!\!\swarrow s'|_{\overline{\alpha}}$.

    With Definition 1.43(vii), we get $s'|_{\overline{\alpha}}\circlearrowleft_F$ (we only added the written field, the fields on the path from $s|_{\overline{os_{0,0}\eta_\alpha}}$ to $s|_{\mathrm{OS}_{0,1}}$, and the fields on the path from $s|_{\mathrm{OS}_{0,0}}$ to $s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}}$ – all these are contained in ρ).

  * We have $\tau\beta \in \mathrm{SPos}(s')$, $\tau'\beta' \notin \mathrm{SPos}(s')$, and $\tau' \in \mathrm{SPos}(s')$. This case is analogous to the previous case.

* We have $\tau\beta \notin \text{SPos}(s')$, $\tau \in \text{SPos}(s')$, $\tau'\beta' \notin \text{SPos}(s')$, and $\tau' \in \text{SPos}(s')$. If $\alpha \lhd \tau\beta$, we have $s|_{\overline{\text{OS}_{0,0}\eta_\alpha}} \leadsto s|_{\text{OS}_{0,1}}$ and $s|_{\text{OS}_{0,0}} \leadsto s|_{\overline{\text{OS}_{0,0}\eta_\alpha}}$, just as in the previous case. Thus, with Definition 1.43(vi, vii), we add $s'|_{\overline{\alpha}} \bigvee s'|_{\overline{\alpha}}$ and $s'|_{\overline{\alpha}} \circlearrowleft_F$ where F is constructed as in the previous case.

  Otherwise, if $\tau\beta \unlhd \alpha$, there is a minimal j with $\tau\, v\, \alpha_1 \dots v\, \alpha_j \notin \text{SPos}(s')$. We then also have $\text{OS}_{0,0}\alpha_j \notin \text{SPos}(s)$ and $s|_{\text{OS}_{0,0}} \leadsto s|_{\overline{\text{OS}_{0,0}\alpha_j}}$. With $c \sqsubseteq s$ and $c|_{\text{OS}_{0,1}} = c|_{\text{OS}_{0,0}\alpha_j}$ we also get $s|_{\overline{\text{OS}_{0,0}\alpha_j}} \leadsto s|_{\text{OS}_{0,1}}$.

  Because of $\tau\beta \unlhd \alpha$ and $\tau\beta \notin \text{SPos}(s')$ we have $s'|_{\overline{\alpha}} = s'|_{\overline{\tau\beta}}$.

  Thus, with Definition 1.43(vi, vii) we get $s'|_{\overline{\alpha}} \bigvee s'|_{\overline{\alpha}}$ and $s'|_{\overline{\alpha}} \circlearrowleft_F$ (we only added the written field, the fields on the path from $s|_{\overline{\text{OS}_{0,0}\alpha_j}}$ to $s|_{\text{OS}_{0,1}}$, and the fields on the path from $s|_{\text{OS}_{0,0}}$ to $s|_{\overline{\text{OS}_{0,0}\alpha_j}}$ – all these are contained in ρ).

* We have $\tau \notin \text{SPos}(s')$ and $\tau'\beta' \in \text{SPos}(s')$. This is not possible, as $\tau = \tau'$.

* We have $\tau\beta \in \text{SPos}(s')$ and $\tau' \notin \text{SPos}(s')$. This is not possible, as $\tau = \tau'$.

* We have $\tau \notin \text{SPos}(s')$, $\tau'\beta' \notin \text{SPos}(s')$, and $\tau' \in \text{SPos}(s')$. This is not possible, as $\tau = \tau'$.

* We have $\tau\beta \notin \text{SPos}(s')$, $\tau \in \text{SPos}(s')$, and $\tau' \notin \text{SPos}(s')$. This is not possible, as $\tau = \tau'$.

* We have $\tau \notin \text{SPos}(s')$ and $\tau' \notin \text{SPos}(s')$. With $c \sqsubseteq s$ and $s|_{\delta(\tau)} = s|_{\text{OS}_{0,1}}$ we know $s|_{\overline{\delta(\tau)}} \bigvee s|_{\text{OS}_{0,1}}$.

  If $\beta = \beta'$, we have $\alpha \rhd \tau\beta = \tau'\beta'$ and $c|_{\text{OS}_{0,0}\eta_\alpha\rho} = c|_{\text{OS}_{0,0}\eta_\alpha\rho'}$ where ρ, ρ' have no common intermediate reference from $\text{OS}_{0,0}\eta_\alpha$ in c. Then, with $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\text{OS}_{0,0}\eta_\alpha\rho} = s|_{\text{OS}_{0,0}\eta_\alpha\rho'}$ or $s|_{\overline{\text{OS}_{0,0}\eta_\alpha}} \bigvee s|_{\overline{\text{OS}_{0,0}\eta_\alpha}}$. Thus, with Definition 1.43(ii,iv) we add $s'|_{\overline{\pi}} \bigvee s'|_{\overline{\pi'}}$. If $\rho' = \varepsilon$, we with Definition 1.10(n) we also have $s|_{\text{OS}_{0,0}\eta_\alpha} = s|_{\text{OS}_{0,0}\eta_\alpha\rho}$ or $s|_{\overline{\text{OS}_{0,0}\eta_\alpha}} \circlearrowleft_F$ where $F \subseteq \rho$. Thus, with Definition 1.43(iii,v) we also add $s'|_{\overline{\pi}} \circlearrowleft_F$.

  Otherwise, if $\beta \neq \beta'$ we know that there is $\gamma \neq \varepsilon$ with $\gamma\, v = \beta$ or $\gamma\, v = \beta'$. Thus, as $c|_{\text{OS}_{0,1}\gamma} = c|_{os_{0,1}}$, with $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\text{OS}_{0,1}} = s|_{\text{OS}_{0,1}\gamma}$ or $s|_{\text{OS}_{0,1}} \bigvee s|_{\text{OS}_{0,1}}$. With Definition 1.43(ii,iv), we also add $s'|_{\overline{\pi}} \bigvee s'|_{\overline{\pi'}}$.

If $\rho' = \varepsilon$, we know $\beta' \lhd \beta$ and $\eta' = \varepsilon$, as ρ, ρ' have no common intermediate reference from α in c'. Thus, $\pi' = \alpha = \tau\beta'$ and $c'|_\alpha = c|_{\mathrm{OS}_{0,0}}$. With $\pi = \tau\beta\eta = \tau\beta'\rho$ we also have $c|_{\mathrm{OS}_{0,0}} = c|_{\mathrm{OS}_{0,0}\rho}$. With $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\mathrm{OS}_{0,0}} = s|_{\mathrm{OS}_{0,0}\rho}$ or $s|_{\mathrm{OS}_{0,0}}\circlearrowleft_F$ with $F \subseteq \rho$. Thus, with Definition 1.43(iii,v) we add $s'|_{\overline{\pi}}\circlearrowleft_F$ with $F \subseteq \rho$.

(o – r) Not applicable, as c' is concrete □

1.6.2. Writing into arrays using AASTORE etc.

Writing into an array is problematic if the necessary information about the array or the used index is not available. In particular, the connection from the array to the reference stored into the array cannot represented in the state without heap predicates.

First, we consider the case when the array we write into does not have any defined index, i.e., if r is the reference of the array and $h(r) = (i_l, f)$ is the data associated with that reference, we have $\mathrm{dom}(f) = \emptyset$.

Let $s \xrightarrow{\text{EVAL}} s'$ be the states involved in the evaluation of a AASTORE operation on an abstract array. An AASTORE decomposition corresponding to the PUTFIELD decomposition in Definition 1.45 is not very useful, as in s' the position corresponding to τv does not exist (as for the array no index is defined). Thus, a case analysis taking into account whether $\tau\beta \in \mathrm{SPOS}(s')$ (for β as in Definition 1.45) is not very useful. Instead, if we write into index i of the array, we define $\beta = i\,\alpha_1\,i \ldots \alpha_n\,i$ and build the proof based on the existence of positions $\mathrm{OS}_{0,0}\alpha_j \in \mathrm{SPOS}(s)$.

Definition 1.46 (Evaluating AASTORE) Let pp_0 be an AASTORE opcode and let s be the corresponding state with $s = (\langle fr_0, \ldots, fr_n\rangle, h, t, hp, sf, e, ic, sr)$ and $fr_i = (pp_i, lv_i, os_i)$. Let $s|_{\mathrm{OS}_{0,0}} =: r_c$, $s|_{\mathrm{OS}_{0,1}} =: i_{\mathsf{index}}$, and $s|_{\mathrm{OS}_{0,2}} =: r_p$. Let there be no reference r with $r =^? r_p$, let there be no heap predicate $r_p?$ and let $(i_l, f) = h(r_p) \in \mathrm{ARRAYS}$ with $\mathrm{dom}(f) = \emptyset$.

Then we define s' with $s \xrightarrow{\text{EVAL}} s'$ as $s' = (\langle fr'_0, fr_1, \ldots, fr_n\rangle, h, t, hp', sf, e, ic, \bot)$ with $fr'_0 = (pp_0, lv_0, os'_0)$. The topmost operand stack os'_0 is identical to os_0, where just the three topmost elements r_c, i_{index}, and r_p are removed. We define $hp' \supseteq hp$ where we add heap predicates according to the following rules:

(i) for all r_b with $r_c \leadsto r_b$ we add $r_p \bigvee\!\!\!\!\!\!\bigvee r_b$ and $r_a \bigvee\!\!\!\!\!\!\bigvee r_b$ for all $r_a \bigvee\!\!\!\!\!\!\bigvee r_p$.

(ii) if there are $\pi, \rho \neq \rho'$ where $s|_\pi = r_c$ and $s|_{\pi\rho} = s|_{\pi\rho'}$, then we add $r_p \bigvee\!\!\!\!\!\!\bigvee r_p$ and $r_a \bigvee\!\!\!\!\!\!\bigvee r_a$ for all $r_a \bigvee\!\!\!\!\!\!\bigvee r_p$.

(iii) if there are π, ρ, ρ' where $s|_\pi = r_c$ and $s|_{\pi\rho} \searrow\!\!\swarrow s|_{\pi\rho'}$, then we add $r_p \searrow\!\!\swarrow r_p$ and $r_a \searrow\!\!\swarrow r_a$ for all $r_a \searrow\!\!\swarrow r_p$.

(iv) if there are $\pi, \rho \neq \rho'$ where $s|_\pi = r_c$ and $s|_{\pi\rho} =^? s|_{\pi\rho'}$, then we add $r_p \searrow\!\!\swarrow r_p$ and $r_a \searrow\!\!\swarrow r_a$ for all $r_a \searrow\!\!\swarrow r_p$.

(v) if there are π, ρ, ρ' with $\rho' \neq \varepsilon$ where $s|_\pi = r_c$ and $s|_{\pi\rho} = s|_{\pi\rho\rho'}$, then we add $r_p \circlearrowleft_F$ and $r_a \circlearrowleft_F$ for all $r_a \searrow\!\!\swarrow r_p$ where F contains all fields in ρ'

(vi) if there are π, ρ with $s|_\pi = r_c$ and $s|_{\pi\rho} \circlearrowleft_F$, then we add $r_p \circlearrowleft_F$ and $r_a \circlearrowleft_F$ for all $r_a \searrow\!\!\swarrow r_p$.

(vii) if there are r_a, r_b with $r_a \rightsquigarrow r_b$, $r_a \rightsquigarrow r_p$, $r_c \rightsquigarrow r_b$, and the paths from r_a to r_p and r_a to r_b have no common intermediate reference, then we add $r_a \searrow\!\!\swarrow r_a$

(viii) if there is r_a with $r_a \rightsquigarrow r_p$, $r_c \rightsquigarrow r_a$ then we add $r_a \circlearrowleft_F$ where F contains the fields on the paths from r_a to r_p and r_c to r_a

As an example, consider that r references an array and we write r into this array (corresponding, for example, to $\mathsf{r[0]} = \mathsf{r}$). Then, even if the state does not explicitly give index information for the array (i.e., $h(r) = (i_l, f)$ with $\mathrm{dom}(f) = \emptyset$), we know that after the write access the written reference r can be reached from the array reference (also r). This implicit information is used in Definition 1.46(viii), where we have $r_a = r_p = r$, $r_c = r_a = r$, and the implicit connection of r_p to r_c using the array index.

Theorem 1.45 Let s, s' be states as in Definition 1.46 with $s \xrightarrow{\text{EVAL}} s'$. Then for all states c, c' with $c \sqsubseteq s$ and $c \xrightarrow{jvm} c'$ we have $c' \sqsubseteq s'$.

As in the proof of Theorem 1.41 and Theorem 1.49, we define a mapping function δ. In this case, δ is very similar to the function defined for the proof of Theorem 1.41, where the topmost operand stack just differs by three instead of two references.

Definition 1.47 (δ) Let $c \xrightarrow{jvm} c'$ be a concrete $\mathsf{AASTORE}$ evaluation with c, c' as defined in Theorem 1.45. Then let δ denote the function that maps positions in c', which has a shorter operand stack than c, to positions in c. For that, let $|\omega| = 1$.

$$\delta(\omega\pi) = \begin{cases} \mathrm{OS}_{0,j+3}\pi & \text{if } \omega = \mathrm{OS}_{0,j} \\ \omega\pi & \text{otherwise} \end{cases}$$

Definition 1.48 (AASTORE decomposition) Let $c \xrightarrow{jvm} c'$ be a concrete AASTORE evaluation with c, c' as defined in Theorem 1.45. Let i be the index used in the write access. For any $\pi \in \text{SPos}(c')$ with $c'|_\pi \neq c|_{\delta(\pi)}$ we define its AASTORE decomposition as $\pi = \tau\beta\eta$ where

- τ is the shortest prefix of π such that both $c'|_\tau = c|_{\text{OS}_{0,2}}$ and $\tau i \trianglelefteq \pi$
- β is the longest position of the form $\beta = i\,\alpha_1\,i\,\alpha_2\,i\,\ldots i\,\alpha_n\,i$ for some $n \geq 0$ where $\tau\beta \trianglelefteq \pi$, $c'|_{\tau i \alpha_j} = c|_{\text{OS}_{0,2}}$, and $c'|_{\tau i \rho} \neq c|_{\text{OS}_{0,2}}$ for all $\varepsilon \neq \rho \lhd \alpha_j$ and all $1 \leq j \leq n$. Note that this implies $c'|_{\tau\beta} = c'|_{\tau i} = c|_{\text{OS}_{0,0}}$ and $c'|_\pi = c|_{\text{OS}_{0,0}\eta}$.

Lemma 1.46 (Change of abstract states by AASTORE) Let s, s', c, c' as defined in Theorem 1.45. For any $\pi \in \text{SPos}(c')$ we have:

- if $c'|_\pi = c|_{\delta(\pi)}$ and $\pi \in \text{SPos}(s')$, then $s'|_\pi = s|_{\delta(\pi)}$
- if $c'|_\pi \neq c|_{\delta(\pi)}$, then for the corresponding AASTORE decomposition $\pi = \tau\beta\eta$ with $\beta = i\,\alpha_1\,i \ldots \alpha_n\,i$ we have
  - $s'|_\tau = s|_{\text{OS}_{0,2}}$ if $\tau \in \text{SPos}(s')$
  - for all $1 \leq j \leq n$ with $\text{OS}_{0,0}\alpha_j \in \text{SPos}(s)$: $s|_{\text{OS}_{0,0}\alpha_j} = s|_{\text{OS}_{0,2}}$
  - $s'|_\pi = s|_{\text{OS}_{0,0}\eta}$ if $\pi \in \text{SPos}(s')$

Proof. According to Definition 1.46 we know $s|_{\text{OS}_{0,2}} \neq \textsf{null}$ and no heap predicate $s|_{\text{OS}_{0,2}}?$ exists, thus with $c \sqsubseteq s$ we have $c|_{\text{OS}_{0,2}} \neq \textsf{null}$. Hence, $c'|_\pi = c|_{\delta(\pi)}$ means that the position π is not influenced by the AASTORE instruction. This implies that we also have $s'|_\pi = s|_{\delta(\pi)}$.

Now let $c'|_\pi \neq c|_{\delta(\pi)}$ and let $\pi = \tau\beta\eta$ be the AASTORE decomposition. Since τ is the shortest prefix of π with $c'|_\tau = c|_{\text{OS}_{0,2}}$ and $\tau\,i \trianglelefteq \pi$, this path is not affected by the evaluation, i.e., $c|_\tau = c|_{\delta(\tau)}$ and $s'|_\tau = s|_{\delta(\tau)}$.

First assume $s'|_\tau \neq s|_{\text{OS}_{0,2}}$. With $c|_{\delta(\tau)} = c'|_\tau = c|_{\text{OS}_{0,2}}$ and $s|_{\delta(\tau)} = s'|_\tau \neq s|_{\text{OS}_{0,2}}$, from $c \sqsubseteq s$ and Definition 1.10(l) we can conclude $s|_{\delta(\tau)} =^? s|_{\text{OS}_{0,2}}$. This contradicts Definition 1.46, where such heap predicates are not allowed. Thus, we have shown that $s'|_\tau = s|_{\text{OS}_{0,2}}$.

Now assume that $s|_{\text{OS}_{0,0}\alpha_j} \neq s|_{\text{OS}_{0,2}}$. We have $c|_{\text{OS}_{0,2}} = c'|_\tau = c'|_{\tau i \alpha_j} = c|_{\text{OS}_{0,0}\alpha_j}$. Thus, $c \sqsubseteq s$ and Definition 1.10(l) implies $s|_{\text{OS}_{0,2}} =^? s|_{\text{OS}_{0,0}\alpha_j}$ or $s|_{\text{OS}_{0,2}} = s|_{\text{OS}_{0,0}\alpha_j}$. Thus, with Definition 1.46 we have $s|_{\text{OS}_{0,0}\alpha_j} = s|_{\text{OS}_{0,2}}$.

As η was not affected by $\mathsf{AASTORE}$, $s'|_{\tau\beta} = s|_{\mathrm{OS}_{0,0}}$ implies $s'|_\pi = s|_{\mathrm{OS}_{0,0}\eta}$. $\square$

Lemma 1.47 Let s, s', c, c', i as defined in Theorem 1.45 and Definition 1.48. Let $\pi \neq \pi' \in \mathrm{SPos}(c')$ with $c'|_\pi = c'|_{\pi'}$, $h_{c'}(c'|_\pi) \in \textsc{Arrays} \cup \textsc{Instances}$, $c'|_\pi \neq c|_{\delta(\pi)}$, $c'|_{\pi'} = c|_{\delta(\pi')}$, and $\pi \notin \mathrm{SPos}(s')$. Let $\pi = \tau\beta\eta$ be the $\mathsf{AASTORE}$ decomposition with $\beta = i\,\alpha_1\,i\,\alpha_2\,i\ldots\alpha_n\,i$. Then we have $s|_{\mathrm{OS0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$.

Proof. If $\delta(\pi'), \mathrm{OS}_{0,0}\eta \in \mathrm{SPos}(s)$, by $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\delta(\pi')} = s|_{\mathrm{OS}_{0,0}\eta}$ or $s|_{\delta(\pi')} =^? s|_{\mathrm{OS}_{0,0}\eta}$. If $\{\delta(\pi'), \mathrm{OS}_{0,0}\eta\} \not\subseteq \mathrm{SPos}(s)$, by $c \sqsubseteq s$ and Definition 1.10(l) we may have $s|_{\overline{\delta(\pi')}} \bigvee s|_{\mathrm{OS}_{0,0}\widetilde{\eta}}$ for some $\widetilde{\eta} \lhd \eta$. If this heap predicate does not exist, we have $s|_{\overline{\delta(\pi')}} = s|_{\overline{\mathrm{OS}_{0,0}\eta}}$. In all cases we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. $\square$

Proof. (of Theorem 1.45 ($\mathsf{AASTORE}$ is correct)) Let c, c', s, s' be states as defined in Theorem 1.45. We show the claim by showing the individual items of Definition 1.10. Let $\pi, \pi' \in \mathrm{SPos}(c')$.

If $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$, the proof is analogous to the corresponding parts of the proof of Theorem 1.41.

Thus, we only consider the cases, where the positions are changed by the $\mathsf{AASTORE}$ operation. According to Lemma 1.46 for $c'|_\pi \neq c|_{\delta(\pi)}$ there is a position η such that $c'|_\pi = c|_{\mathrm{OS}_{0,0}\eta}$ and $s'|_\pi = s|_{\mathrm{OS}_{0,0}\eta}$. Similarly, for $c'|_{\pi'} \neq c|_{\delta(\pi')}$ there is a position η' such that $c'|_{\pi'} = c|_{\mathrm{OS}_{0,0}\eta'}$ and $s'|_{\pi'} = s|_{\mathrm{OS}_{0,0}\eta'}$.

(a – c) Trivial.

(d – l) These cases do not need to be considered, as $\pi \in \mathrm{SPos}(s')$ and $c'|_\pi \neq c|_{\delta(\pi)}$ contradict each other (π traverses through the array, but the array does not represent any index).

(m) Let $\pi \neq \pi'$, $c'|_\pi = c'|_{\pi'}$ with $h_{c'}(c'|_\pi) \in \textsc{Instances} \cup \textsc{Arrays}$. W.l.o.g. assume $\pi \notin \mathrm{SPos}(s')$. We handle both the cases that $\pi' \in \mathrm{SPos}(s')$ and $\pi' \notin \mathrm{SPos}(s')$ here.

- Assume we have $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. Let $\pi = \tau\beta\eta$ with $\beta = i\,\alpha_1\,i\ldots i\,\alpha_n\,i$ be the $\mathsf{AASTORE}$ decomposition.
  - We have $\tau \in \mathrm{SPos}(s')$. With Lemma 1.47 we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. Thus, Definition 1.46(i) requires $s'|_{\overline{\tau}} \bigvee s'|_{\overline{\pi'}}$.

- We have $\tau \notin \mathrm{SPos}(s')$. Then also $\delta(\tau) \notin \mathrm{SPos}(s)$ and as $c|_{\delta(\tau)} = c|_{\mathrm{OS}_{0,2}}$ and $c \sqsubseteq s$, with Definition 1.10(m) we have $s|_{\overline{\delta(\tau)}} \,\vee\!\!\!\!\!\;\vee\, s|_{\mathrm{OS}_{0,2}}$. With Lemma 1.47 we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. Thus, Definition 1.46(i) requires $s'|_{\overline{\tau}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$.

• Assume we have $c'|_{\pi} = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. This case is analogous to the previous case.

• Assume we have $c'|_{\pi} \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. Let $\pi = \tau\beta\eta$ and $\pi' = \tau'\beta'\eta'$ be the AASTORE decompositions with $\beta = i\,\alpha_1\, i \ldots \alpha_n\, i$ and $\beta' = i\,\alpha'_1\, i \ldots \alpha'_{n'}\, i$.

- We have $\tau \in \mathrm{SPos}(s')$ and $\tau' \in \mathrm{SPos}(s')$. If there is a j with $1 \leq j \leq n$ and $\mathrm{OS}_{0,0}\alpha_j \notin \mathrm{SPos}(s)$, then with $c|_{\mathrm{OS}_{0,2}} = c|_{\mathrm{OS}_{0,0}\alpha_j}$ and $c \sqsubseteq s$, with Definition 1.10(m) we have $s|_{\mathrm{OS}_{0,2}} \,\vee\!\!\!\!\!\;\vee\, s|_{\mathrm{OS}_{0,0}\widetilde{\alpha_j}}$ for some $\widetilde{\alpha_j} \trianglelefteq \alpha_j$. Similarly, if there is a j' with $1 \leq j' \leq n'$ and $\mathrm{OS}_{0,0}\alpha'_{j'} \notin \mathrm{SPos}(s)$, we have $s|_{\mathrm{OS}_{0,2}} \,\vee\!\!\!\!\!\;\vee\, s|_{\mathrm{OS}_{0,0}\widetilde{\alpha'_{j'}}}$ for some $\widetilde{\alpha'_{j'}} \trianglelefteq \alpha'_{j'}$. Thus, according to Definition 1.46(i) we add $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$.

  Otherwise, for all $1 \leq j \leq n$ and $1 \leq j' \leq n'$ we have $\mathrm{OS}_{0,0}\alpha_j, \mathrm{OS}_{0,0}\alpha'_{j'} \in \mathrm{SPos}(s)$.

  If $\beta \neq \beta'$ we know that there is $\rho \neq \varepsilon$ with $\rho i = \beta$ or $\rho i = \beta'$. Thus, as $c|_{\mathrm{OS}_{0,2}\rho} = c|_{\mathrm{OS}_{0,2}}$ with $c \sqsubseteq s$ we have $s|_{\mathrm{OS}_{0,2}} = s|_{\mathrm{OS}_{0,2}\rho}$ or $s|_{\mathrm{OS}_{0,2}} \,\vee\!\!\!\!\!\;\vee\, s|_{\mathrm{OS}_{0,2}}$. Hence, with Definition 1.46(ii) we have $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$.

  Otherwise, if $\beta = \beta'$, we may have $\{\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta'\} \not\subseteq \mathrm{SPos}(s)$ and $s|_{\overline{\mathrm{OS}_{0,0}\eta}} \,\vee\!\!\!\!\!\;\vee\, s|_{\overline{\mathrm{OS}_{0,0}\eta'}}$. If so, according to Definition 1.46(iii) we add $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$. If this heap predicate does not exist, we know that $s|_{\overline{\mathrm{OS}_{0,0}\eta}} = s|_{\overline{\mathrm{OS}_{0,0}\eta'}}$ where $\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta'$ have the same suffix w.r.t. s. As $\beta = \beta'$, we do not need to add $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$.

  Finally, we consider the remaining case that $\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta' \in \mathrm{SPos}(s)$. With $c \sqsubseteq s$ we have $s|_{\mathrm{OS}_{0,0}\eta} = s|_{\mathrm{OS}_{0,0}\eta'}$ or $s|_{\mathrm{OS}_{0,0}\eta} =^? s|_{\mathrm{OS}_{0,0}\eta'}$. In the first case, if $\eta = \eta'$, we do not need to add a joins heap predicate (as $\beta = \beta'$). If $\eta \neq \eta'$ with Definition 1.46(ii) we add $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$. In the latter case, with Definition 1.46(iv) we add $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$.

- We have $\{\tau, \tau'\} \not\subseteq \mathrm{SPos}(s')$. Thus, with $c \sqsubseteq s$ and Definition 1.10(m) we may have $s|_{\overline{\delta(\tau)}} \,\vee\!\!\!\!\!\;\vee\, s|_{\overline{\delta(\tau')}}$ and hence, $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$. If this joins heap predicate does not exist, we know $s|_{\overline{\delta(\tau)}} = s|_{\overline{\delta(\tau')}}$ where $\delta(\tau), \delta(\tau')$ have the same suffix w.r.t. s, thus we also have $s'|_{\overline{\tau}} = s'|_{\overline{\tau'}}$ where τ, τ' have the same suffix w.r.t. s'. If $\beta = \beta'$ and $\eta = \eta'$, we do not need to have $s'|_{\overline{\pi}} \,\vee\!\!\!\!\!\;\vee\, s'|_{\overline{\pi'}}$.

Otherwise, with $c \sqsubseteq s$ and $s|_{\delta(\tau)} = s|_{\mathrm{OS}_{0,2}}$ we know $s|_{\overline{\delta(\tau)}} \curlyvee s|_{\mathrm{OS}_{0,2}}$. If $\beta \neq \beta'$ we know that there is $\rho \neq \varepsilon$ with $i\,\rho = \beta$ or $i\,\rho = \beta'$. Thus, as $c|_{\mathrm{OS}_{0,0}\rho\, i} = c|_{os_{0,0}}$, with $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\mathrm{OS}_{0,0}} \curlyvee s|_{\mathrm{OS}_{0,0}}$. With Definition 1.43(iii), we also add $s'|_{\overline{\pi}} \curlyvee s'|_{\overline{\pi'}}$.

If $\beta = \beta'$ and $\eta \neq \eta'$, with $c|_{\mathrm{OS}_{0,0}\eta} = c|_{\mathrm{OS}_{0,0}\eta'}$, $c \sqsubseteq s$, and Definition 1.10(n) we have $s|_{\mathrm{OS}_{0,0}\eta} = s|_{\mathrm{OS}_{0,0}\eta'}$ or $s|_{\overline{\mathrm{OS}_{0,0}\alpha}} \curlyvee s|_{\overline{\mathrm{OS}_{0,0}\alpha}}$ for some α. Thus, with Definition 1.43(ii,iii) we add $s'|_{\overline{\pi}} \curlyvee s'|_{\overline{\pi'}}$.

(n) Let $\pi = \alpha\rho, \pi' = \alpha\rho'$ where ρ, ρ' have no common intermediate reference from α in c' and let $\rho \neq \varepsilon$. Let $c'|_\pi = c'|_{\pi'}$. We have $\{\pi, \pi'\} \not\subseteq \mathrm{SPos}(s')$. W.l.o.g. assume $\pi \notin \mathrm{SPos}(s')$. We handle both the cases that $\pi' \in \mathrm{SPos}(s')$ and $\pi \notin \mathrm{SPos}(s')$ here. In all cases we need to show $s'|_{\overline{\alpha}} \curlyvee s'|_{\overline{\alpha}}$ and, if $\rho' = \varepsilon$, also $s'|_{\overline{\alpha}} \circlearrowleft_F$ with $F \subseteq \rho$.

If $c'|_\pi \neq c|_{\delta(\pi)}$, let $\pi = \tau\beta\eta$ be the $\mathsf{AASTORE}$ decomposition with $\beta = i\,\alpha_1\, i\,\alpha_2\, i \ldots \alpha_n\, i$ Similarly, if $c'|_{\pi'} \neq c|_{\delta(\pi')}$ let $\pi' = \tau'\beta'\eta'$ and $\beta' = i\,\alpha'_1\, i\,\alpha'_2\, i \ldots \alpha'_{n'}\, i$.

- Assume we have $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} = c|_{\delta(\pi')}$. Hence, with $c \sqsubseteq s$, we have $s|_{\overline{\delta(\alpha)}} \rightsquigarrow s|_{\mathrm{OS}_{0,2}}$. Similarly, we have $s|_{\overline{\delta(\alpha)}} \rightsquigarrow s|_{\overline{\delta(\pi')}} = s|_{\overline{\delta(\alpha\rho')}}$. With Lemma 1.47 we also have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi')}}$. Thus, Definition 1.46(vii) requires $s'|_{\overline{\alpha}} \curlyvee s'|_{\overline{\alpha}}$.

  If $\rho' = \varepsilon$, we have $s|_{\overline{\delta(\alpha)}} = s|_{\overline{\delta(\pi')}}$. Thus, we have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\alpha)}}$. According to Definition 1.46(viii) we then have $s'|_{\overline{\alpha}} \circlearrowleft_F$ with $F \subseteq \rho$ (we only added the fields on the path from $s|_{\overline{\delta(\alpha)}}$ to $s|_{\mathrm{OS}_{0,2}}$ and the fields on the path from $s|_{\mathrm{OS}_{0,0}}$ to $s|_{\overline{\delta(\pi')}}$ – all these are contained in ρ).

- Assume we have $c'|_\pi = c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. Hence, with $c \sqsubseteq s$, we have $s|_{\overline{\delta(\alpha)}} \rightsquigarrow s|_{\mathrm{OS}_{0,2}}$. Similarly, we have $s|_{\overline{\delta(\alpha)}} \rightsquigarrow s|_{\overline{\delta(\pi)}} = s|_{\overline{\delta(\alpha\rho)}}$. With Lemma 1.47 we also have $s|_{\mathrm{OS}_{0,0}} \rightsquigarrow s|_{\overline{\delta(\pi)}}$. Thus, Definition 1.46(vii) requires $s'|_{\overline{\alpha}} \curlyvee s'|_{\overline{\alpha}}$.

  If $\rho' = \varepsilon$, we have $c|_\alpha = c|_{\delta(\pi)}$. With $c \sqsubseteq s$, Definition 1.10(n), $s|_{\overline{\delta(\alpha)}} = s'|_{\overline{\alpha}}$, and $s|_{\overline{\delta(\pi)}} = s'|_{\overline{\pi}}$, we do not need to add a heap predicate.

- Assume we have $c'|_\pi \neq c|_{\delta(\pi)}$ and $c'|_{\pi'} \neq c|_{\delta(\pi')}$. As ρ, ρ' have no common intermediate reference, and both π, π' are affected by the $\mathsf{AASTORE}$ operation, we conclude $\tau = \tau \lhd \alpha$. Let $\alpha = \tau\beta_\alpha\eta_\alpha$ be the $\mathsf{AASTORE}$ decomposition, thus $c|_{\mathrm{OS}_{0,0}\eta_\alpha} = c'|_\alpha$.

  – We have $\beta = \beta'$. Thus, we have $c|_{\mathrm{OS}_{0,0}\eta} = c|_{\mathrm{OS}_{0,0}\eta'}$ with $\eta \neq \eta'$.

    If $\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta' \in \mathrm{SPos}(s)$, with Definition 1.46(ii) we add $s'|_{\overline{\pi}} \curlyvee s'|_{\overline{\pi'}}$. If also $\rho' = \varepsilon$, with Definition 1.46(v) we also add $s'|_{\overline{\pi}} \circlearrowleft_F$ with $F \subseteq \rho$.

Otherwise, if $\{\mathrm{OS}_{0,0}\eta, \mathrm{OS}_{0,0}\eta'\} \not\subseteq \mathrm{SPos}(s)$, with $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}} \searrow\!\!\swarrow s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}}$ and, if $\rho' = \varepsilon$, $s|_{\overline{\mathrm{OS}_{0,0}\eta_\alpha}}\circlearrowleft_F$ with $F \subseteq \rho$. Thus, with Definition 1.46(iii,vi) we add $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$ and, if $\rho' = \varepsilon$, also $s'|_{\overline{\pi}}\circlearrowleft_F$.

– We have $\beta \neq \beta'$. We know that there is $\gamma \neq \varepsilon$ with $\gamma\, i = \beta$ or $\gamma\, i = \beta'$. Thus, as $c|_{\mathrm{OS}_{0,2}\gamma} = c|_{\mathrm{OS}_{0,2}}$, with $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\mathrm{OS}_{0,2}} = s|_{\mathrm{OS}_{0,2}\gamma}$ or $s|_{\mathrm{OS}_{0,2}} \searrow\!\!\swarrow s|_{\mathrm{OS}_{0,2}}$. With Definition 1.46(ii,iii), we also add $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.

  If $\rho' = \varepsilon$, we know $\beta' \lhd \beta$ and $\eta' = \varepsilon$, as ρ, ρ' have no common intermediate reference from α in c. Thus, $\pi' = \alpha = \tau\beta'$ and $c'|_\alpha = c|_{\mathrm{OS}_{0,0}}$. With $\pi = \tau\beta\eta = \tau\beta'\rho$ we also have $c|_{\mathrm{OS}_{0,0}} = c|_{\mathrm{OS}_{0,0}\rho}$. With $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\mathrm{OS}_{0,0}} = s|_{\mathrm{OS}_{0,0}\rho}$ or $s|_{\mathrm{OS}_{0,0}}\circlearrowleft_F$ with $F \subseteq \rho$. Thus, with Definition 1.46(v,vi) we add $s'|_{\overline{\pi}}\circlearrowleft_F$ with $F \subseteq \rho$.

(o – r) Not applicable, as c' is concrete □

Definition 1.46 can only be used to evaluate a AASTORE operation if the array does not define any index, i.e., for $h(r) = (i_l, f)$ we have $\mathrm{dom}(f) = \emptyset$. However, we may have a situation where $\mathrm{dom}(f) \neq \emptyset$, but the index i used to write into the array is unknown. As the size of the array is known to be of a specific size an integer refinement on the index variable could be used in order to result in situations where for both the array and the index all information needed for a (trivial) evaluation is available.

As arrays tend to be very large (and, thus, the index refinement may produce many states), instead in the implementation, we automatically transform arrays with $\mathrm{dom}(f) \neq \emptyset$ into an array with $\mathrm{dom}(f) = \emptyset$. This is done by replacing the old array (with $\mathrm{dom}(f) \neq \emptyset$) by a new and empty array (with $\mathrm{dom}(f') = \emptyset$) and then storing all previously stored elements (in f) into the new array (in f') using the procedure outlined in Definition 1.46. As this is rather technical, in this thesis we will not provide further details for this workaround.

1.6.3. Reading from arrays using AALOAD etc.

Loading from an array using one of the opcodes IALOAD, LALOAD, FALOAD, DALOAD, AALOAD, BALOAD, CALOAD, or SALOAD is trivial in a state where the array index is known and where the content of the array is represented (i.e., if for an index i_1 with $h(i_1) = [i, i] \in \textsc{Integers}$ we have $h(o) = (i_l, f) \in \textsc{Arrays}$ with $i \in \mathrm{dom}(f)$).

In all other cases, we can add a new reference as we did in Definition 1.22, with the only difference that we do not alter the array content function f. However, in the case of an AALOAD opcode, we need to model the relationship between the added reference and the other references on the heap.

In the following definition we only consider the case that we actually read from an array, i.e., no exception is thrown (the array exists and the index is in the bounds defined by the array size). Furthermore, we demand that no $=^?$ heap predicate exists for the array reference. This is no real restriction, as equality refinement can be used to remove such heap predicates.

Definition 1.49 (Evaluating AALOAD) Let s be a state evaluating AALOAD, reading from an array $r = s|_{\mathrm{OS}_{0,1}}$ at index $i_{\mathsf{index}} = s|_{\mathrm{OS}_{0,0}}$. Let there be no reference $\hat{r}$ with $r =^? \hat{r}$ and let $h(r) = (i_l, f) \in \textsc{Arrays}$. Let i_{index} be a reference with $h(i_{\mathsf{index}}) \in \textsc{Integers}$. Furthermore, we demand that $|h(i_{\mathsf{index}})| \geq 2$ or $|h(i_l)| \geq 2$ (i.e., i_{index} or i_l do not reference a literal value).

Then we define s' with $s \xrightarrow{\textsc{Eval}} s'$ as $s' = (\langle fr'_0, fr_1, \ldots, fr_n \rangle, h', t', hp', sf, e, ic, \bot)$. We introduce a new reference r' that is the value read from the array. In s' we define $fr'_0 = (pp_0, lv_0, os'_0)$. The topmost operand stack os'_0 is identical to os_0, where the two topmost elements r and i are removed and the reference r' is added at position $\mathrm{OS}_{0,0}$.

Let the component type componenttype be some class, interface, or array. Let $T_{\mathsf{componenttype}} \subseteq \textsc{Types}$ be the abstract type that contains exactly all arrays and non-abstract classes which are subtypes of componenttype. As in Definition 1.22, if $T_{\mathsf{componenttype}} = \emptyset$ we (re)define $r' = \mathsf{null}$, $t' = t + \{r' \mapsto \emptyset\}$ and disregard the following definitions of s'.

Otherwise, $T_{\mathsf{componenttype}} \neq \emptyset$ and we define s' using

- $h' = h + \{r' \mapsto f' \in \textsc{Instances}\}$ where $\mathrm{dom}(f') = \emptyset$
- $t' = t + \{r' \mapsto T_{\mathsf{componenttype}}\}$

Furthermore, we extend the heap predicates as follows:

$$
\begin{aligned}
hp' = hp & \\
&\cup \{r' =^? r'' \mid \exists j \in \mathbb{N} : s|_{\mathrm{OS}_{0,1j}} = r''\} && (1)\\
&\cup \{r' =^? r'' \mid \exists j \in \mathbb{N} : s|_{\mathrm{OS}_{0,1j}} = r_\gamma \wedge r'' =^? r_\gamma\} && (2)\\
&\cup \{r' \searrow\!\swarrow r'' \mid \exists \tau : s|_{\mathrm{OS}_{0,1\tau}} = r_\gamma \wedge r'' \searrow\!\swarrow r_\gamma\} && (3)\\
&\cup \{r' \searrow\!\swarrow r\} && (4)\\
&\cup \{r' \searrow\!\swarrow r'' \mid \exists \tau \neq \varepsilon, j \in \mathbb{N} : s|_{\mathrm{OS}_{0,1j\tau}} = r_\gamma \wedge r'' =^? r_\gamma\} && (5)\\
&\cup \{r' \searrow\!\swarrow r'' \mid \exists \tau \neq \varepsilon, j \in \mathbb{N} : s|_{\mathrm{OS}_{0,1j\tau}} = r''\} && (6)\\
&\cup \{r' \searrow\!\swarrow r' \mid \exists \tau : s|_{\mathrm{OS}_{0,1\tau}} = r_\gamma \wedge r_\gamma \searrow\!\swarrow r_\gamma\} && (7)\\
&\cup \{r' \circlearrowleft_F \mid \exists \tau : s|_{\mathrm{OS}_{0,1\tau}} = r_\gamma \wedge r_\gamma \circlearrowleft_F\} && (8)\\
&\cup \{r' =^? r'' \mid r \searrow\!\swarrow r''\} && (9)\\
&\cup \{r'?\}
\end{aligned}
$$

As in the proof for Theorem 1.41, for the correctness proof we need a function δ that maps positions from c to c'.

Definition 1.50 ($\boldsymbol{\delta}$) Let $c \xrightarrow{jvm} c'$ be a concrete AALOAD evaluation with c, c' as defined in Theorem 1.49. For $h_c(c|_{\mathrm{OS}_{0,0}}) = [i, i] \in \textsc{Integers}$ let $i \in \mathbb{N}$ be the array index we read from.

Then let δ denote the partial function that maps positions in c', which has a shorter operand stack than c, to positions in c. For that, let $|\omega| = 1$.

$$\delta(\omega\pi) = \begin{cases} \mathrm{OS}_{0,j+1}\pi & \text{if } \omega = \mathrm{OS}_{0,j} \wedge j > 0 \\ \mathrm{OS}_{0,1}\, i\pi & \text{if } \omega = \mathrm{OS}_{0,0} \\ \omega\pi & \text{otherwise} \end{cases}$$

Lemma 1.48 Let s, s', c, c' as in Definition 1.50. If $\pi \neq \mathrm{OS}_{0,0}$ and $\pi \in \textsc{SPos}(s')$, then $\delta(\pi) \in \textsc{SPos}(s)$.

Proof. When evaluating an AALOAD opcode, we drop two references from the operand stack (the array read from, and the index) and put a reference onto the operand stack (the value read from the array). We know that $\{\mathrm{OS}_{0,0}\tau\} \cap \textsc{SPos}(s') = \{\mathrm{OS}_{0,0}\}$. Thus, as $\pi \neq \mathrm{OS}_{0,0}$, π references data that also is available in s. The corresponding mapping is given using δ in Definition 1.50. □

Theorem 1.49 Let s, s' be a states as defined in Definition 1.49, i.e., $s \xrightarrow{\textsc{Eval}} s'$. Let c, c' be concretes state with $c \sqsubseteq s$ and $c \xrightarrow{jvm} c'$. Then we also have $c' \sqsubseteq s'$.

Proof. Let s, s', c, c', i, Ψ as in Definition 1.50, Theorem 1.49, and Lemma 1.48. Let $\Psi = \{\mathrm{OS}_{0,0}\tau \in \textsc{SPos}(c')\}$ be the positions at or below the read reference on the operand stack in c'. For all positions $\pi \in \textsc{SPos}(s')$, if $\delta(\pi) \in \textsc{SPos}(s)$, we have $s|_{\delta(\pi)} = s'|_\pi$.

We prove $c' \sqsubseteq s'$ by checking all conditions of Definition 1.10. Let $\pi, \pi' \in \textsc{SPos}(c')$. W.l.o.g. we restrict to positions $\pi \in \Psi$. For this note that the values at $\textsc{SPos}(s)$ are left unchanged by the read access, i.e., we have $s|_{\delta(\pi)} = s'|_\pi$ for all $\pi \in \textsc{SPos}(s)$, and all heap predicates from s also exist in s'. Thus, with $c \sqsubseteq s$, most conditions of Definition 1.10 are trivially satisfied. Furthermore, the properties for the new reference provided at position $\mathrm{OS}_{0,0}$ also correspond to Definition 1.10, as already shown in the proof of Theorem 1.16.

To summarize, in this proof we concentrate on the relationships of references on the heap, where we only consider connections that somehow involve the reference read from the array.

(a–k) Trivial (cf. proof of Theorem 1.16).

(l) If $c'|_{\pi} = c'|_{\pi'}$, $\pi \neq \pi'$, $h_{c'}(c'|_{\pi}) \in \textsc{Instances} \cup \textsc{Arrays}$, and $\pi, \pi' \in \mathrm{SP}\textsc{os}(s')$, then we have $\pi = \mathrm{OS}_{0,0}$. We also know $\delta(\pi') \in \mathrm{SP}\textsc{os}(s)$, as $\pi' \neq \mathrm{OS}_{0,0}$ and $\pi' \in \mathrm{SP}\textsc{os}(s')$ holds.

If $\delta(\pi) = \mathrm{OS}_{0,1}\, i \in \mathrm{SP}\textsc{os}(s)$, then with $c \sqsubseteq s$ and Definition 1.10(l) we have $s|_{\mathrm{OS}_{0,1}\, i} = s|_{\delta(\pi')}$ or $s|_{\mathrm{OS}_{0,1}\, i} =^? s|_{\delta(\pi')}$. Thus, according to rules 1 and 2 in Definition 1.49 we also introduced the heap predicate $s'|_{\mathrm{OS}_{0,0}} =^? s'|_{\pi'}$.

Otherwise, if $\mathrm{OS}_{0,1}\, i \notin \mathrm{SP}\textsc{os}(s)$, then $\overline{\mathrm{OS}_{0,1}\, i}_s = \mathrm{OS}_{0,1}$ and with $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\mathrm{OS}_{0,1}} \searrow\!\!\swarrow s|_{\delta(\pi')}$ (as $\delta(\pi) \notin \mathrm{SP}\textsc{os}(s)$, but $\delta(\pi') \in \mathrm{SP}\textsc{os}(s)$). Thus, according to rule 9 in Definition 1.49 we also introduced the heap predicate $s'|_{\mathrm{OS}_{0,0}} =^? s'|_{\pi'}$.

(m) If $c'|_{\pi} = c'|_{\pi'}$, $\pi \neq \pi'$, $h_{c'}(c'|_{\pi}) \in \textsc{Instances} \cup \textsc{Arrays}$, and $\{\pi, \pi'\} \not\subseteq \mathrm{SP}\textsc{os}(s')$, then one of the following cases holds.

- $\pi = \mathrm{OS}_{0,0} \in \mathrm{SP}\textsc{os}(s'), \pi' \notin \mathrm{SP}\textsc{os}(s')$, and
  - $\delta(\pi) \in \mathrm{SP}\textsc{os}(s)$. Thus, with $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\delta(\pi)} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$, as $\delta(\pi) \in \mathrm{SP}\textsc{os}(s)$, but $\delta(\pi') \notin \mathrm{SP}\textsc{os}(s)$. Thus, according to rule 3 in Definition 1.49 we added $s'|_{\pi} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$.
  - $\delta(\pi) \notin \mathrm{SP}\textsc{os}(s)$, thus $\overline{\delta(\pi)}_s = \mathrm{OS}_{0,1}$. Thus, with $c \sqsubseteq s$ and Definition 1.10(m) we may have $s|_{\overline{\delta(\pi)}} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$. If so, according to rule 3 in Definition 1.49 we added $s'|_{\pi} \searrow\!\!\swarrow s'|_{\overline{\pi'}}$. If we do not have $s|_{\overline{\delta(\pi)}} \searrow\!\!\swarrow s|_{\overline{\delta(\pi')}}$, we know $s|_{\overline{\delta(\pi)}} = s|_{\overline{\delta(\pi')}}$. With $\overline{\delta(\pi)}_s = \mathrm{OS}_{0,1}$, according to rule 4 in Definition 1.49 we added $s'|_{\overline{\pi'}} \searrow\!\!\swarrow s'|_{\pi}$.
- $\pi \notin \mathrm{SP}\textsc{os}(s'), \pi' \in \mathrm{SP}\textsc{os}(s')$.
  - If $\pi' \in \Psi$, we have $\pi' = \mathrm{OS}_{0,0}$. This case is analogous to the first case.
  - If $\pi' \notin \Psi$, according to Lemma 1.48 we have $\delta(\pi') \in \mathrm{SP}\textsc{os}(s)$. With $c \sqsubseteq s$ and Definition 1.10(l,m) we have $s|_{\overline{\delta(\pi)}} \searrow\!\!\swarrow s|_{\delta(\pi')}$ or $s|_{\overline{\delta(\pi)}} =^? s|_{\delta(\pi')}$. Thus, according to rules 3 and 5 in Definition 1.49 we have $s'|_{\overline{\pi}} \searrow\!\!\swarrow s'|_{\pi'}$.
- $\pi \notin \mathrm{SP}\textsc{os}(s'), \pi' \notin \mathrm{SP}\textsc{os}(s')$

- $\delta(\overline{\pi}_{s'}) \in \text{SPos}(s)$
  - $\pi' \notin \Psi$. Then with $c \sqsubseteq s$ and Definition 1.10(m) we may have $s|_{\overline{\delta(\pi)}} \bigvee\!\!\!\!\swarrow s|_{\overline{\delta(\pi')}}$. If so, according to rule 3 we then also have $s'|_{\overline{\pi}} \bigvee\!\!\!\!\swarrow s'|_{\overline{\pi'}}$. If we do not have $s|_{\overline{\delta(\pi)}} \bigvee\!\!\!\!\swarrow s|_{\overline{\delta(\pi')}}$, then we know $s|_{\overline{\delta(\pi)}} = s|_{\overline{\delta(\pi')}}$. With $\pi \in \Psi$ we know that there is a connection from $s|_{\text{OS}_{0,0}\,i}$ to $s|_{\overline{\delta(\pi')}}$, i.e., there exists τ with $s|_{\text{OS}_{0,0}\,i\tau} = s|_{\overline{\delta(\pi')}}$. Thus, with rule 6 we add $s'|_{\overline{\pi}} \bigvee\!\!\!\!\swarrow s'|_{\overline{\pi'}}$.
  - $\pi' \in \Psi$. We have $\overline{\pi}_{s'} = \overline{\pi'}_{s'} = \text{OS}_{0,1}$. Thus, we need $s'|_{\text{OS}_{0,0}} \bigvee\!\!\!\!\swarrow s'|_{\text{OS}_{0,0}}$. With $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\text{OS}_{0,1}\tau} \bigvee\!\!\!\!\swarrow s|_{\text{OS}_{0,1}\tau}$ for some τ. Thus, according to rule 7 we add the necessary heap predicate.
- $\delta(\overline{\pi}_{s'}) \notin \text{SPos}(s)$. Thus, $\overline{\delta(\pi)} = \text{OS}_{0,1}$.
  - $\pi' \notin \Psi$. Thus, with $c \sqsubseteq s$ and Definition 1.10(m) we have $s|_{\overline{\delta(\pi)}} \bigvee\!\!\!\!\swarrow s|_{\overline{\delta(\pi')}}$. According to rule 3 we add $s|_{\overline{\pi}} \bigvee\!\!\!\!\swarrow s|_{\overline{\pi'}}$.
  - $\pi' \in \Psi$. Thus, we have $\overline{\delta(\pi)}_s = \overline{\delta(\pi')}_s = \text{OS}_{0,1}$ and with $c \sqsubseteq s$ and Definition 1.10(n) we have $s|_{\overline{\delta(\pi)}} \bigvee\!\!\!\!\swarrow s|_{\overline{\delta(\pi')}}$. According to rule 7 we add $s|_{\overline{\pi}} \bigvee\!\!\!\!\swarrow s|_{\overline{\pi'}}$.

(n) Let $\pi = \alpha\tau$ and $\pi' = \alpha\tau'$ with $\tau \neq \varepsilon$, τ, τ' have no common intermediate reference from α in c', $h_{c'}(c'|_{\pi}) \in \text{Instances} \cup \text{Arrays}$, and $c'|_{\pi} = c'|_{\pi'}$. Then we have $\{\pi, \pi'\} \not\subseteq \text{SPos}(s')$ and $\alpha = \text{OS}_{0,0}\rho$ for some ρ. With $c \sqsubseteq s$ and Definition 1.10(n) we then have $s|_{\overline{\delta(\alpha)}} \circlearrowleft_F$ and $s|_{\overline{\delta(\alpha)}} \bigvee\!\!\!\!\swarrow s|_{\overline{\delta(\alpha)}}$. Thus, with rules 8 and 7 we then also have $s'|_{\overline{\alpha}} \circlearrowleft_F$ and $s'|_{\overline{\alpha}} \bigvee\!\!\!\!\swarrow s'|_{\overline{\alpha}}$.

(o – r) Not applicable, as c is concrete. □

1.6.4. Class instances and interned Strings

As explained at the beginning of this chapter, the technique presented in this thesis does not correctly handle java.lang.Class object instances resulting out of calls to the opcode corresponding to x.class or using the native method x.getClass().

Example 1.50

```
public void test(Object x) {
  Object y = new Object();
  if (x == y.class) {
     // may be true
  }
  Object z = new Object();
  if (x == z.getClass()) {
    // may be true
  }
}
```

Similarly, instances of java.lang.String created from constant strings or using the native method x.intern() may also be identical to already existing String object instances.

Example 1.51

```
public void test(Object x) {
  String y = new String("abc");
  if (x == y.intern()) {
     // may be true
  }

  // constant strings are automatically "interned"
  // i.e., "abc" == "abc".intern() is guaranteed
  String z = "abc";
  if (x == z) {
    // may be true
  }
}
```

With the following changes to the technique presented in this thesis, handling of Class and String object instances can be done according to [GJS+12]. Whenever an object instance of type Class or String is created using the mentioned concepts, we just add the heap predicates $=^?$ to connect the created object with all objects that may be the same object instance.

However, it may also be the case that an object instance, e.g. a list, has a successor that is the created Class or String object instance. If the created object did not exist at another position in the state before, it is not possible to denote this sharing using the presented heap predicates. Instead, when creating a new object instance of Class or String we add $\searrow\!\!\swarrow$ heap predicates to all already existing object instances that may have a successor that is the created object instance. Note that this implicit sharing also demands minor changes to the intersection process (Definition 1.38), which we will not present here.

In order to also represent handling of Class and String object instances using concrete states (i.e., without heap predicates), we would need to extend the state definition to also contain two maps. One map assigns a Class object instance to each class. The other map assigns an object instance of type String to each interned character sequence. With this new definition of states all concepts working on states, most prominently the instance definition (Definition 1.10) also need to be adapted.

Note that there may be situations where getClass() or intern() is invoked on an object instance with an unknown type or a String object representing an unknown character sequence, respectively. In this case the maps mentioned above do not help and, as a consequence, we need to add heap predicates.

1.7. Abstraction

In order to obtain a finite Symbolic Execution Graph even for infinite computations with an infinite number of states, we need to introduce abstraction. As recursion is not considered in this chapter, any nonterminating program run must traverse a loop in the program infinitely often. Thus, there is an opcode in the program which appears in infinitely many states of the computation. In order to introduce abstraction, before we continue evaluation with a state, we look for similar states with the same opcode in the graph. If such a state is found, we introduce abstraction based on the information of the two states.

Let s_1, s_2 be states in the graph with the same opcodes, initialization status, and exception status. Let s_{start} be the start state of the graph. If there are paths in the graph from $s|_{\mathsf{start}}$ to s_1 and from s_1 to s_2, i.e., s_2 comes after s_1, we demand that $s_2 \sqsubseteq s_1$ or enforce abstraction. By this abstraction as described in Section 2.2 we obtain a state s_3 with $s_1 \sqsubseteq s_3$ and $s_2 \sqsubseteq s_3$. As $s_2 \sqsubseteq s_3$ and $s_2 \not\sqsubseteq s_1$, with Theorem 1.4 this gives us $s_2 \sqsubseteq s_3 \not\sqsubseteq s_1$, thus also $s_3 \neq s_1$ and $s_1 \sqsubsetneq s_3$. Thus, when we add s_3 to the graph, we add a state with *strictly less information* than s_1. By considering both the information from s_1 and s_2 this process quickly reaches a suitably abstract state that represents all possible computations of any given loop. Note that $s_2 = s_3$ is allowed and not problematic, as still we have $s_1 \sqsubsetneq s_2 = s_3$.

Thus, when we have found a suitably abstract state s_n, for the next repeating state s_{n+1} we can make use of the fact that all computations represented by s_{n+1} are already represented by s_n and connect s_{n+1} back to s_n using an *instance edge*, denoted $s_{n+1} \xrightarrow{\text{INSTANCE}} s_n$ and forming a cycle in the Symbolic Execution Graph.

However, so far we did not show that starting from any state there are only finitely many abstraction steps possible. If this was not the case, the idea explained above could still lead to a non-terminating graph construction (where we abstract infinitely often).

Theorem 1.52 (Finite Abstraction Height) Let s_0 be a state. Then there is no infinite sequence of states $s_0 \sqsubsetneq s_1 \sqsubsetneq s_2 \cdots$.

Proof. Assume there is an infinite sequence of states $s_0 \sqsubsetneq s_1 \sqsubsetneq s_2 \sqsubsetneq \cdots$. Consider Definition 1.10(a–r). For $x \in \{a, \ldots, r\}$ let $s_0^x \sqsubsetneq s_1^x \sqsubsetneq s_2^x \cdots$ be the states from that sequence in which the corresponding information was abstracted. As an example, if $x = i$ we know that for $s_j^i \sqsubsetneq s_{i+1}^i$ there is a position π_j with $\mathrm{dom}(h_{s_j^i}(s_j^i|_{\pi_j})) \supsetneq \mathrm{dom}(h_{s_{i+1}^i}(s_{j+1}^i|_{\pi_j}))$ (i.e., in each step we lost information about at least one field). In the case of heap predicates, adding any heap predicate introduces abstraction.

We now consider each item of Definition 1.10, except f (dealing with integers) and g (dealing with types), and show that the corresponding sequence $s_0^x \sqsubsetneq s_1^x \sqsubsetneq \ldots$ must be finite. It suffices to consider positions that exist in both states, as $\textsc{SPos}(s) \subseteq \textsc{SPos}(s')$ for $s' \sqsubseteq s$ (by Lemma 1.3).

(a–d) No abstraction is possible.

(e) For each position only a single abstraction step is possible.

(h) For each position only a single abstraction step is possible.

(i) As each type only has a finite number of instance fields, for each reference only a finite number of abstraction steps is possible.

(j) Only two abstraction steps are possible.

(k) No abstraction step is possible.

(l) Only a single abstraction step is possible.

(m–r) As there are only a finite number of references in each state, only a finite number of heap predicates can be added to the state. In the case of $\circlearrowleft_F$, there are only finitely many instance fields known in the program, thus also only finitely many subsets exist.

In the case of Definition 1.10(f) an infinite sequence as described above is possible. For example, we could have the integer information $[5,5] \subsetneq [5,6] \subsetneq [5,7] \subsetneq \cdots$. However, in our implementation we avoid this problem by using a counter for each value in Integers. As an example, if $[5,5]$ with counter c_1 is abstracted to $[5,6]$ with counter c_2, we demand $c_2 > c_1$. If we want to abstract a value with a counter above a certain threshold, we only allow abstraction to $(-\infty, \infty)$. Thus, by using a counter for values from Integers as described above, we also cannot have an infinite number of abstraction steps involving Definition 1.10(f).

Finally, we may have an infinite sequence involving Definition 1.10(g). While $|2^{\mathbb{N}\times(\textsc{PrimTypes}\cup\textsc{ClassNames})}|$ is infinite, when analyzing Java Bytecode programs we only encounter a finite subset. As both PrimTypes and ClassNames are finite, we only need to regard arrays. Using Java Bytecode it is possible to create multi-dimensional arrays. However, the dimension of each created array is a constant defined in the program, thus it may not be provided for example using an integer variable containing an arbitrary number.

As there is no $x \in \{a, \ldots, r\}$ with an infinite sequence $s_0^x \subsetneq s_1^x \subsetneq \cdots$, the sequence $s_0 \subsetneq s_1 \subsetneq s_2 \subsetneq \ldots$ also cannot be infinite. □

Note that it is not necessary to introduce abstraction for every repetition as described above. Instead, it suffices to abstract after the opcode was repeated for an arbitrary, but finite, number of times. However, in our implementation, we abstract after the first repetition.

1.8. Symbolic Execution Graphs

Using the concepts of refinement, abstract evaluation, and abstraction we can construct a Symbolic Execution Graph for a given Java Bytecode program. However, so far we did not define what exactly a Symbolic Execution Graph is and what its properties are.

Definition 1.51 (Symbolic Execution Graph) For a given start state $s \in \textsc{States}$, the corresponding Symbolic Execution Graph is a graph $\mathcal{G} = (N, E, \mathcal{L})$ with $N \subseteq \textsc{States}$ and $E \subseteq N \times N$. The labelling function $\mathcal{L}$ with $\mathrm{dom}(\mathcal{L}) = E$ gives additional information for each edge in the graph. First, it defines the type of each edge, which either is an evaluation edge, a refinement edge, or an instance edge. For evaluation edges, the label may provide additional information that can be used in further analyses. For example, in the construction of term rewrite systems, relations of the form $i_1 \leq i_2$

are added to an evaluation edge if the relation was important for the evaluation of the opcode (e.g., due to a comparison, or due to an array access where we know that the index is within the bounds of the array).

Furthermore, we impose restrictions on $\mathcal{G}$. $\mathcal{G}$ only is a Symbolic Execution Graph for the start state s if

(i) it has finitely many states

(ii) the only state without incoming edges is the start state

(iii) all states with no outgoing edge have an empty call stack

(iv) each state with an evaluation successor has no other successor

(v) each state with a refine successor only has refine successors

(vi) each state with an instance successor only has instance successors

(vii) every loop in the graph contains at least one $\xrightarrow{\textsc{Eval}}$ edge

Using the techniques presented in this thesis, for any start state $s \in \textsc{States}$ we are able to construct a corresponding Symbolic Execution Graph as described in Algorithm 2 (which was already shown on page 31).

Algorithm 2: Graph construction

Input: $s_0 \in \textsc{States}$
Output: Symbolic Execution Graph $\mathcal{G}$
initialize $\mathcal{G}$
$\textsc{AddState}(\mathcal{G}, s_0)$
for all $s \in \textsc{LeafStates}(\mathcal{G})$ **do**
    if s is a repetition of s' **then**
        if $s \sqsubseteq s'$ **then**
            connect s to s' using an instance edge
        else
            $\textsc{ForceAbstraction}(s, s')$
    else
        if s can be evaluated **then**
            $\textsc{Evaluate}(s)$
        else
            $\textsc{Refine}(s)$

LeafStates The method $\textsc{LeafStates}$ returns all states without any outgoing edge and which have a non-empty call stack. These are the states that still need to be evaluated, as no program end is reached and no other state exists of which it is an instance.

Refine The method REFINE refines the state. In Section 1.4 we present several refinement techniques that can be used to refine the information of a state so that, after a finite number of refinement steps, evaluation of the resulting state(s) is possible. The resulting states are added to the graph and connected to the state provided as input using refinement edges.

Evaluate Similarly, the method EVALUATE evaluates based on the given information. The details for most opcodes are not given explicitly in this thesis. However, after applying refinement if necessary, evaluation is straightforward for most opcodes. In the case of PUTFIELD, AALOAD, and AASTORE detailed evaluation information is given. For details we refer to Section 1.6. The resulting state is added to the graph and connected to the state provided as input using an evaluation edge.

repetitions When finding out whether s is a repetition of s', we only consider a state s' if it does not have an outgoing instance edge, it has no incoming refinement edge, and there is a path from s_0 to s' to s in the graph. The state s then is a repetition of s' if both states have the same *shape*. This includes all opcodes, all return addresses, the set exception, and the initialization state (Definition 1.10(a–c)). We also demand that in s' no split result is set, i.e., the corresponding component is set to $\perp$.

ForceAbstraction The method FORCEABSTRACTION calls Algorithm 3 (cf. Section 2.2) and adds the resulting state to the graph. The input states are connected to the resulting state using *instance edges*. If one of the input states already had an outgoing non-instance edge, this edge and all states and edges which then are not reachable from the start state are deleted from the graph.

We now show that this algorithm indeed terminates and provides a corresponding Symbolic Execution Graph.

Lemma 1.53 Let $s \in$ STATES. When applied to s, Algorithm 1 terminates.

Proof. Assume that all invoked methods terminate (we provide additional details in Chapter 2). Then the algorithm does not terminate if either a leaf is handled infinitely often, or an infinite number of leaves is added to the graph. The only case where no state is added, is line 6. Here, an outgoing edge is added for a leaf, thus the number of leaves strictly decreases.

In EVALUATE and REFINE new leaves are added and are connected to other states of the graph. Thus, if this happens infinitely often, we obtain an infinite sequence of (different) states in the graph.

By construction, each sequence of refinement edges is finite. Next, we consider sequences of evaluation edges. If there is a sequence of different states containing an infinite number of evaluation edges, the sequence contains two states s_i, s_j of the same shape (as we are only considering non-recursive programs in this chapter). For this it is important to know that the number of opcodes and, therefore, return addresses in any program is finite. Furthermore, as the number of known classes is finite, the number of different initialization states also is finite. Thus, the sequence cannot be infinite as we close a loop (in line 6), or we obtain an infinite sequence of $\sqsubsetneq$ edges. According to Theorem 1.52 this also is not possible.

Finally, we regard the abstraction process in FORCEABSTRACTION. Here, note that we only remove outgoing edges of states if we add a new instance edge. Thus, no new leaves are created. □

Lemma 1.54 (Refinement Edges) For each state s with an outgoing refinement edge we also have that s only has outgoing refinement edges.

Let $s_i \xrightarrow{\text{REFINE}} s_{i+1} \xrightarrow{\text{REFINE}} s_{i_2} \cdots$ be a sequence of refinement edges in the graph. If the construction is completed, there is a s_n with $s_i \xrightarrow{\text{REFINE}} s_{i+1} \xrightarrow{\text{REFINE}} s_{i_2} \cdots s_{n-1} \xrightarrow{\text{EVAL}} s_n$. If the construction is still running, the last state in the sequence is a leaf.

In both cases in the sequence each state only appears once.

Proof. According to Algorithm 2 we only add refinement edges as outgoing edges of a leaf. Thus, as long as no other edge is added to a non-leaf state, each state with a single outgoing refinement edge only has outgoing refinement edges. We only add edges to non-leaf states in line 6. However, here we also remove all outgoing refinement edges. Thus, each state s with an outgoing refinement edge only has outgoing refinement edges.

By construction we know that after finitely many refinement steps we reach a state that can be evaluated. The state at the end of the refinement sequence cannot be a repeating state, as the state at the start of the refinement sequence also is not repeating. Thus, as we only add refinement edges pointing to leaves, each sequence of refinement edges either ends in a leaf or with an evaluation edge.

By construction we also know that different states are created in the refinement process (where at least the split result differs). Thus, in the sequence each state is only contained once. □

Theorem 1.55 Let $s \in \text{STATES}$. When applied to s, Algorithm 1 terminates and produces a Symbolic Execution Graph with start state s.

Proof. According to Lemma 1.53 we know that the graph construction terminates, i.e., we get a graph $\mathcal{G} = (N, E, \mathcal{L})$. We need to show that $\mathcal{G}$ actually is a Symbolic Execution Graph.

(i) As the construction terminates, N is finite.

(ii) For every added state, except the start state, we also add an incoming edge. Furthermore, if we remove an incoming edge, we also remove the state or ensure that the state has another incoming edge.

(iii) The algorithm only terminates if no state without outgoing edge and with non-empty call stack exists.

(iv) In line 6 we add an outgoing instance edge to a leaf. In FORCEABSTRACTION we only add instance edges to states that only have outgoing instance edges (if not, other edges are removed). In EVALUATE and REFINE we only add edges to leaves.

(v) The claim is shown in Lemma 1.54.

(vi) In line 6 we add an outgoing instance edge to a leaf. In FORCEABSTRACTION we remove non-instance edges.

(vii) Let $s_i \to s_{i+1} \to \cdots s_n \to s_i$ be a loop without evaluation edges, where the states $s_i, \ldots, s_n$ are different.

The edge $s_n \to s_i$ can only result out of line 6, as in all other cases we only add incoming edges to a leaf and s_i already has an outgoing edge. Then we know that s_n is a repetition of s_i. Thus, we also know that s_i does not have an outgoing instance edge and also does not have an incoming refinement edge. Thus, as s_n has a non-empty call stack, s_i either has at least one outgoing refinement edge or a single outgoing evaluation edge ($s_i \to s_{i+1}$). As shown in Lemma 1.54, at the end of each refinement sequence there is an evaluation edge.

Thus, the claim is shown. □

A Symbolic Execution Graph corresponding to this definition has the interesting property that for each concrete evaluation sequence starting in a state represented by the start state of the graph there is a corresponding path in the graph. As there are only evaluation steps in the concrete evaluation sequence, but in the graph we may need to use refinement or abstraction, there may be several states in the graph that correspond to a single state in the concrete evaluation sequence.

Theorem 1.56 Let $c \xrightarrow{jvm} c'$ be a concrete computation step. If a Symbolic Execution Graph $\mathcal{G}$ contains a state s^1 with $c \sqsubseteq s^1$ (where s^1 may be the start state of the graph), then there also is a sequence $s^1, \dots, s^n, s'$ of states contained in $\mathcal{G}$ with the following properties:

- if $n > 1$ there is a refine or instance edge $s^{i-1} \to s^i$ for each $1 < i \leq n$
- there is an evaluation edge $s^n \xrightarrow{\text{EVAL}} s'$ with $c' \sqsubseteq s'$
- we have $c \sqsubseteq s^i$ for all $1 \leq i \leq n$

Proof. Let c, c', s^1 as in Theorem 1.56. As $c \xrightarrow{jvm} c'$, we know that c does not have an empty call stack. With $c \sqsubseteq s^1$ the same holds for s^1. Thus, with Definition 1.51 we know that there is a successor state of s^1.

If there is an evaluation edge $s^1 \xrightarrow{\text{EVAL}} s'$ with Theorem 1.38 we know that $c' \sqsubseteq s'$. Otherwise, if s^1 has an outgoing refinement edge, we know there is an edge $s^1 \xrightarrow{\text{REFINE}} s^2$ with $c \sqsubseteq s^2$ (as every refinement is valid). With Lemma 1.54 we also know that each refinement sequence ends with $s^n \xrightarrow{\text{EVAL}} s'$ where $c \sqsubseteq s^n$ and $c' \sqsubseteq s'$. If we have $s^1 \xrightarrow{\text{INSTANCE}} s^2$, we have $s^1 \sqsubseteq s^2$, thus, with Theorem 1.4, also $c \sqsubseteq s^2$. As each loop in the graph contains at least one evaluation edge and both s^1 and s^2 have a non-empty call stack, we know that after a finite sequence of instance edges we have $s^n \xrightarrow{\text{EVAL}} s'$ with $c \sqsubseteq s^n$ and $c' \sqsubseteq s'$. $\square$

Based on Theorem 1.56 we can draw several conclusions.

Corollary 1.57 Let $c_1, c_2, \dots$ be a finite or infinite concrete computation sequence. Let $\mathcal{G}$ be a Symbolic Execution Graph with start state s where $c_1 \sqsubseteq s$. Then the graph contains a sequence of states $s_1^1, \dots, s_1^{n_1}, s_2^1, \dots s_2^{n_2}, s_3^1, \dots$ with the following properties:

- if $n_i > 1$ there is a refine or instance edge $s_i^{j-1} \to s_i^j$ for each $1 < j \leq n_i$
- there is an evaluation edge $s_i^{n_i} \xrightarrow{\text{EVAL}} s_{i+1}^1$ with $c_{i+1} \sqsubseteq s_{i+1}^1$
- we have $c_i \sqsubseteq s_i^j$ for all $1 \leq j \leq n_i$

Corollary 1.58 Let $c_1, c_2, \ldots$ be a finite or infinite concrete computation sequence. Let $\mathcal{G}$ be a Symbolic Execution Graph with start state s where $c_1 \sqsubseteq s$. Then for each c_i the graph contains a state s_i such that $c_i \sqsubseteq s_i$.

Corollary 1.59 Let $\mathcal{G}$ be a Symbolic Execution Graph with start state s. Let c be a concrete state. If for all states s_i contained in the graph we have $c \not\sqsubseteq s_i$, then there is no concrete evaluation sequence containing c and starting in any state c_i with $c_i \sqsubseteq s$.

Corollary 1.60 Let $\mathcal{G}$ be a Symbolic Execution Graph with start state s. If for any infinite path of states $s_1^1, \ldots, s_1^{n_1}, s_2^1, \ldots, s_2^{n_2}, \ldots$ as described in Theorem 1.56 there is no concrete computation sequence $c_1, c_2, \ldots$ with $c_1 \sqsubseteq s$ and $c_i \sqsubseteq s_i^j$ for all $i, 1 \leq j \leq n_i$, then all concrete computation sequences starting in a concrete state c with $c \sqsubseteq s$ are finite.

1.9. Conclusion and Outlook

With the results presented in Section 1.8, Symbolic Execution Graphs may be used for several interesting further analyses. As already discussed, it is possible to prove termination of a program by constructing a corresponding Symbolic Execution Graph and then showing that for each infinite path in the graph (i.e., a loop is traversed infinitely often), there cannot be a corresponding concrete computation sequence. For further details we refer to the PhD thesis of Marc Brockschmidt [Bro14].

Furthermore, the information contained in Symbolic Execution Graphs constructed as described in this thesis is very precise. Thus, although an overapproximation of all possible computation sequences is represented, this approximation is quite precise. In Chapter 4 we present a technique that uses the information provided in a Symbolic Execution Graph and identifies parts of the code that do not have any influence to some intended result defined by the user. Thus, Symbolic Execution Graphs may be used to optimize programs and/or find bugs that are hard to find using existing tools.

The technique presented in this chapter is not able to deal with recursive programs. This limitation is addressed in Chapter 3.

Extending this approach by better shape analysis is an interesting topic for future work. For example, currently abstraction of doubly linked lists (i.e., linked lists with an additional pointer to the previous list element) is far from perfect, as the structure of such lists cannot completely be described using the heap predicates presented in this chapter. As a result, for algorithms working on doubly linked lists, unnecessary sharing and cyclicity effects are introduced during graph construction.

Furthermore, one should think about possible use cases for the presented analysis. While it is possible to create Symbolic Execution Graphs containing very precise information, this comes at the cost of performance. If this precision is important, for example to find certain hard to find bugs as explained in Chapter 4, this may be reasonable. However, if termination analysis of real world JAVA programs is the goal, more sophisticated abstraction heuristics must be found. In addition, extending the analysis to be able to analyze only parts of a program and re-using previously computed analysis (i.e., modularization) is a very interesting topic for future work.

2. Automation

While Chapter 1 mostly dealt with the theoretical aspects of constructing Symbolic Execution Graphs, when implementing the techniques, one faces several technical challenges.

One of these challenges is to develop software based on the formal specifications so that the resulting product can be extended over the course of several years. Development of the AProVE project started in 2001 and until now about 550,000 source lines of code have been developed. Of those, about 52,000 are directly related to the analysis of JAVA BYTECODE. As the intricacies of developing software such as AProVE are detached from the specific area of software verification, we will not go into further details. Instead we just emphasize that, in addition to the theoretical results, the individual techniques have been implemented in AProVE. Furthermore, as the approach presented in Chapter 1 allows for different levels of abstraction, heuristics need to be applied when and how state information is abstracted. We will not present the details of the heuristics implemented in AProVE. Instead, we refer to [Bro14] in which the techniques presented in this thesis are extended to analyze the termination behavior of JAVA programs. In [Bro14] the author also provides experiments comparing the power of AProVE with competing tools in the area of automated termination analysis of JAVA programs.

In this chapter the focus lies on algorithmic solutions to problems for which a direct adaption of the already presented formalizations is not easily possible. Here the most difficult aspect is how infinite data structures are handled, where the definition of state positions (cf. Definition 1.5) is of special interest. For a state s containing cycles, the set $\text{SPos}(s)$ contains an infinite number of state positions. As such, especially aspects of the instance definition (cf. Definition 1.10) which uses statements such as "For all $\pi, \pi' \in \text{SPos}(s')$" are hard to automate.

2.1. Abstract Types

We start with issues related to the representation of abstract types as in Definition 1.1. The range $2^{\mathbb{N} \times (\text{PrimTypes} \cup \text{ClassNames})}$ of TYPES is infinite[1] and, thus, handling abstract types in an implementation is not straightforward. As we observed that abstract types of the form "packageName.ClassName or any subtype" are used quite frequently, we de-

[1] In fact, an array has at most 255 dimensions. However, even with this limitation a naive implementation would not work in practice.

cided to make use of the notion of "or any subtype" in combination with the possibility to express types without also meaning their subtypes. Using the suffix "|" we denote concrete types, whereas the suffix "..." is used to denote the mentioned type and all its subtypes. For example, the set described by {java.lang.Object...} both contains java.lang.Object and java.lang.String, but also [I denoting an array of type int (as every array is an object). So, in our implementation, the type information really corresponds to a function $\textsc{References} \rightarrow 2^{\mathbb{N}\times(\textsc{PrimTypes} \,\cup\, \textsc{ClassNames})\times\{|,...\}}$. By also allowing entries X... for interfaces X we can often find short representations of the types used in the construction of the Symbolic Execution Graph.

In our implementation we answer the question "is type X contained in the abstract type" by first looking for an entry X| or X.... If this is not contained, we *expand* all entries by adding Z... for all subtypes Z of Y if Y... is contained in the abstract type. We repeat this process as often as necessary, which is determined by height of the type tree. For example, if we want to find out if the type Z is contained in an abstract type {java.lang.Object...} where we have Z extends Y and Y extends java.lang.Object, we expand twice: first we expand {java.lang.Object...} to {Y...}, then we expand to {Z...}.

To also correctly deal with arrays, in this expansion step we also expand java.lang.Object... to [java.lang.Object... and [P for all $P \in \textsc{PrimTypes}$[2]. This expansion process is repeated according to the array dimension of X, i.e., to find [[java.lang.String in the abstract type {java.lang.Object...}, we first expand giving us [java.lang.Object..., then expand again to obtain [[java.lang.Object... and finally expand a third time to obtain [[java.lang.String....

This idea is also used for other common operations on abstract types, such as intersection or subset inclusion (as used in Definition 1.10).

2.2. Merge

For ForceAbstraction in Algorithm 1 we need an algorithm that allows us to *merge* the information of two states s_1, s_2 with $s_2 \not\sqsubseteq s_1$ into a state s_3 such that $s_1 \sqsubsetneq s_3$ and $s_2 \sqsubseteq s_3$ hold. We only call this algorithm with input states that have the same shape, i.e., the call stacks have the same height, they contain the same opcodes and return addresses, both states have the same initialization information, and both or none of the two states have a set exception reference.

As we are interested in keeping as much information as possible, we design the algorithm to keep all information that is present in both states, while allowing all values which are allowed by any of the input states. This can be seen as the counterpart of the intersection process presented in Section 1.5, where we retain information as long as it is represented in at least one of the two input states.

[2]As arrays also implement the interfaces java.io.Serializable and java.lang.Cloneable, we also expand those to array types if corresponding entries exist in the abstract type

As an example, consider the situation that we have $s_2 \not\sqsubseteq s_1$ for two nearly identical states. Let the only difference be that in s_2 two local variables may contain the same object as indicated with the heap predicate $r_1 =^? r_2$, while in s_1 we know that the referenced objects are different. Then, we could create the merge result state s_3 as a copy of s_2 with $s_2 \sqsubseteq s_3$ and $s_1 \sqsubsetneq s_3$.

In other situations, if we do not have $s_1 \sqsubseteq s_2$, we still create a state that keeps as much information as possible. For example, if for both s_1 and s_2 we know that a local variable is null, this information should also be represented in s_3 and there is no need to have a more abstract value for that local variable.

In the following algorithm we consider this idea by traversing both input states simultaneously and storing information in s_3 that is present in both states. If there is data where one of the states has more abstract information, it is stored in s_3. If neither of the information is more abstract than the other (for example as for some local variable in s_1 we have a null, while in s_2 we have an existing object instance), we create more abstract information based on the inputs (in the example, conforming to Definition 1.10(h), the heap predicate $r?$ would be used, and the field information of the object instance would be abstracted, i.e., $\mathrm{dom}(f) = \emptyset$).

Algorithm 3 is the Merge algorithm, which traverses the input states and creates the merged state. We will show an example of how this algorithm is used in Section 2.2.1. We first create an empty state and then set the initialization state of s_1, which is identical to the information in s_2. Then, as we know that either both states have no exception reference or an exception is set in both states, we set the exception component of the resulting state accordingly. Here, in the case that exception references exist in the two input states, the algorithm MergeReferences (shown in Algorithm 4) is used to compute the reference used in the resulting state.

In the remainder of the Merge algorithm, the references stored in the static fields and the call stack are merged likewise. When traversing the call stack, we make use of the fact that both input states have the same call stack height. As we only analyze verified Java Bytecode and the two states have the same opcode, we know that the corresponding stack frames have the same number of local variables and entries on the operand stack. Finally, after all references are set in the state, the heap predicates are added using various algorithms.

The task of the algorithm MergeReferences, as shown in Algorithm 4, is not only to return a reference that can be used in the resulting state, but also to add the referenced data to the state. For this, depending on the type of the referenced data, we might need to abstract information. Thus, MergeReferences first analyzes the referenced data and then calls a specialized algorithm. As all return addresses in the input states are identical, any return address found is returned unchanged. For primitive types, we use MergePrimitives. While this algorithm is not shown, it is trivial for floating

Algorithm 3: MERGE

Input: $s_1 = (\langle fr^1_0, \ldots, fr^1_{n_1}\rangle, h_1, t_1, hp_1, sf_1, e_1, ic_1, \bot)$ with $fr^1_i = (pp^1_i, lv^1_i, os^1_i)$,
$s_2 = (\langle fr^2_0, \ldots, fr^2_{n_2}\rangle, h_2, t_2, hp_2, sf_2, e_2, ic_2, \bot)$ with $fr^2_i = (pp^2_i, lv^2_i, os^2_i)$

Output: $s_3 = (\langle fr^3_0, \ldots, fr^3_{n_3}\rangle, h_3, t_3, hp_3, sf_3, e_3, ic_3, \bot)$ with $fr^3_i = (pp^3_i, lv^3_i, os^3_i)$

create state s_3 with call stack of height n_1, empty heap, no local variable, ...
$ic_3 := ic_1$
if $e_1 = \bot \wedge e_2 = \bot$ **then**
  $e_3 := \bot$
else
  $e_3 :=$ MERGEREFERENCES(e_1, e_2)

for all $v \in \mathrm{dom}(sf_1)$ **do**
  $sf_3(v) :=$ MERGEREFERENCES$(sf_1(v), sf_2(v))$

for all $0 \le i \le n_1$ **do**
  for all $0 \le j < \mathrm{sizeof}(lv^1_i)$ **do**
    $lv^3_i(j) :=$ MERGEREFERENCES$(lv^1_i(j), lv^2_i(j))$
  for all $0 \le j < \mathrm{sizeof}(os^1_i)$ **do**
    $os^3_i(j) :=$ MERGEREFERENCES$(os^1_i(j), os^2_i(j))$
  $pp^3_i := pp^1_i$

MERGECYCLICPREDICATES() // Definition 1.10(o), see Algorithm 8
MERGEPOSSIBLEEXISTENCE() // Definition 1.10(p), see Algorithm 9
MERGEPOSSIBLEEQUALITY() // Definition 1.10(q), see Algorithm 10
MERGEJOINSPREDICATES() // Definition 1.10(r), see Algorithm 11

ADDNEWPOSSIBLEEQUALITY() // Definition 1.10(l), see Algorithm 12
ADDNEWJOINSPREDICATES() // Definition 1.10(m), see Algorithm 13
IDENTIFYNONTREESHAPES() // Definition 1.10(n), see Algorithm 14

point values (which can only be abstracted to $\perp$). For integer values the task also is simple, where attention needs to be given to disallow infinite abstraction as explained in Section 1.7. In all other cases, one of the algorithms MergeNull, MergeArrays, and MergeInstances is used to do the actual computation.

There may be cyclic references on the heap. Thus, when traversing the heap and creating the merged values, we may need to merge the same pair of references more than once (in the case of cyclic data structures possibly even infinitely often). In order to have a terminating algorithm, we remember the resulting reference for each merged pair of references using SetMerged in the individual merge algorithms, before traversing into the data structures. This information is used in lines 1–2 of MergeReferences, so that the problem hinted at above does not occur.

Finally, the type of non-primitive values is computed using MergeTypes, which is not shown. Instead, we refer to Section 2.1.

Algorithm 4: MergeReferences

Input: $r_1, r_2 \in$ References
Output: $r_3 \in$ References
1: **if** MergedTo(r_1, r_2, r_3) **then**
2:    **return** r_3
3: **if** r_1 is a return address **then**
4:    **return** r_1
5: **if** $h_1(r_1) \in$ Integers $\cup$ Floats **then**
6:    **return** MergePrimitives(r_1, r_2)
7: **if** $r_1 = \mathsf{null} \vee r_2 = \mathsf{null}$ **then**
8:    $r_3 :=$ MergeNull(r_1, r_2)
9: **if** $h_1(r_1) \in$ Arrays $\vee\ h_2(r_2) \in$ Arrays **then**
10:    $r_3 :=$ MergeArrays(r_1, r_2)
11: **else**
12:    $r_3 :=$ MergeInstances(r_1, r_2)
13: $t_3(r_3) =$ MergeTypes$(t_1(r_1), t_2(r_2))$
14: **return** r_3

In Algorithm 5 we show MergeNull. We need to make sure that when setting r? for some reference the referenced data contains no field information (line 7).

In Algorithm 6 we have MergeArrays. As arrays only are instances of objects without field information, when merging an array with some instance, for the resulting instance we do not define any field (lines 3, 5). We respect the restriction that for concrete arrays either no index is known or we have some information for each index. Thus, if we know that the array length is a literal in the merged state (i.e., it also is the same literal in the input states), we retain information about the content of the array in lines 10–12.

In the case of instances, with Algorithm 7 (MergeInstances), we retain information for each field set in both input object instances.

Algorithm 5: MergeNull

Input: $r_1, r_2 \in \textsc{References}$
Output: $r_3 \in \textsc{References}$

if $r_1 = \mathsf{null} \land r_2 = \mathsf{null}$ **then**
    return null
else
    create fresh reference r_3
    $\textsc{SetMerged}(r_1, r_2, r_3)$
    add r_3? to hp_3
    $f_3 := ()$ // i.e., $\mathrm{dom}(f_3) = \emptyset$
    $h_3(r_3) := f_3 \in \textsc{Instances}$
    return r_3

Algorithm 6: MergeArrays

Input: $r_1, r_2 \in \textsc{References}$
Output: $r_3 \in \textsc{References}$

create fresh reference r_3
$\textsc{SetMerged}(r_1, r_2, r_3)$
$f_3 := ()$ // i.e., $\mathrm{dom}(f_3) = \emptyset$
if $h_1(r_1) \in \textsc{Instances} \lor h_2(r_2) \in \textsc{Instances}$ **then**
    $h_3(r_3) := f_3 \in \textsc{Instances}$
    return r_3
$(i_l^1, f_1) := h_1(r_1)$
$(i_l^2, f_2) := h_2(r_2)$
$i_l^3 := \textsc{MergePrimitives}(i_l^1, i_l^2)$
if $h_3(i_l^3)$ is a literal **then**
    for all $i \in \mathrm{dom}(f_1)$ **do**
        $f_3(i) := \textsc{MergeReferences}(f_1(i), f_2(i))$
$h_3(r_3) := (i_l^3, f_3) \in \textsc{Arrays}$
return r_3

Algorithm 7: MergeInstances

Input: $r_1, r_2 \in \textsc{References}$
Output: $r_3 \in \textsc{References}$

create fresh reference r_3
$\textsc{SetMerged}(r_1, r_2, r_3)$
$f_3 := ()$ // i.e., $\mathrm{dom}(f_3) = \emptyset$
for all $v \in \mathrm{dom}(h_1(r_1)) \cap \mathrm{dom}(h_2(r_2))$ **do**
    $f_3(v) := \textsc{MergeReferences}(h_1(r_1)(v), h_2(r_2)(v))$
$h_3(r_3) := f_3 \in \textsc{Instances}$
return r_3

The following seven algorithms all are used to add heap predicates to the merged state. While Algorithms 8 to 11 copy over already existing heap predicates and are rather simple, Algorithms 12 to 14 introduce new heap predicates and are rather complicated. In all of these algorithms we deal with information from both s_1 and s_2 in a very similar manner. Thus, in the representation as shown in this thesis, large parts of the algorithms are copied, where the only changes are related to addressing s_2 instead of s_1. These copied parts are shown in this smaller and less prominent font.

In Algorithm 8 (MergeCyclicPredicates) we deal with Definition 1.10(o) and add $\circlearrowleft_F$ heap predicates to s_3 if a corresponding heap predicate exists in s_1 or s_2. According to Definition 1.10(o), we also need to add heap predicates if the corresponding positions do not exist in s_3 (as indicated by $\overline{\pi}_{s_3}$ in the definition). For that we make use of RealizedPositions. The call RealizedPositions(r_1, s_1, s_3) gives the references in s_3 which are at a position $\overline{\pi}_{s_3}$, where we have $s_1|_\pi = r_1$. As we may have infinitely many positions π with $s_1|_\pi = r_1$, this is not trivial. In Section 2.3 we explain how RealizedPositions can be implemented.

In line 3 of the algorithm we may add $r_3\circlearrowleft_F$ to hp_3 even though we already have $r_3\circlearrowleft_{F'}$. In this case we intersect the field sets, i.e., we remove $r_3\circlearrowleft_{F'}$ and instead add $r_3\circlearrowleft_{F\cap F'}$.

Algorithm 8: MergeCyclicPredicates

```
1: for all r₁↻F ∈ hp₁ do
2:    for all r₃ ∈ RealizedPositions(r₁, s₁, s₃) do
3:       add r₃↻F to hp₃
4: for all r₂↻F ∈ hp₂ do
5:    for all r₃ ∈ RealizedPositions(r₂, s₂, s₃) do
6:       add r₃↻F to hp₃
```

Algorithm 9 shows MergePossibleExistence. Here, we add the $r?$ heap predicate to the corresponding references in s_3. In the algorithms called by MergeReferences, we store which pairs of references are merged into which resulting references. This information can be accessed by MergedTo and helps us identify which corresponding references are used in the merged state. For example, if we merged $(r_a, r_b) \mapsto r_c$, in line 9 of Algorithm 9 this helps us to obtain r_a when we have r_b and r_c.

Algorithm 9: MergePossibleExistence

```
1: for all r₁? ∈ hp₁ do
2:    for all r₂, r₃ : MergedTo(r₁, r₂, r₃) do
3:       add r₃? to hp₃
4: for all r₂? ∈ hp₂ do
5:    for all r₁, r₃ : MergedTo(r₁, r₂, r₃) do
6:       add r₃? to hp₃
```

In Algorithms 10 and 11 (MergePossibleEquality and MergeJoinsPredicates) we again make use of MergedTo and RealizedPositions.

Algorithm 10: MergePossibleEquality

for all $r_1 =^? r_1' \in hp_1$ **do**
  for all r_2, r_3 : $\textsc{MergedTo}(r_1, r_2, r_3)$ **do**
    for all r_2', r_3' : $\textsc{MergedTo}(r_1', r_2', r_3')$ **do**
      add $r_3 =^? r_3'$ to hp_3
for all $r_2 =^? r_2' \in hp_2$ **do**
  for all r_1, r_3 : $\textsc{MergedTo}(r_1, r_2, r_3)$ **do**
    for all r_1', r_3' : $\textsc{MergedTo}(r_1', r_2', r_3')$ **do**
      add $r_3 =^? r_3'$ to hp_3

Algorithm 11: MergeJoinsPredicates

for all $r_1 \searrow\!\!\swarrow r_1' \in hp_1$ **do**
  for all $r_3 \in \textsc{RealizedPositions}(r_1, s_1, s_3)$ **do**
    for all $r_3' \in \textsc{RealizedPositions}(r_1', s_1, s_3)$ **do**
      add $r_3 \searrow\!\!\swarrow r_3'$ to hp_3
for all $r_2 \searrow\!\!\swarrow r_2' \in hp_2$ **do**
  for all $r_3 \in \textsc{RealizedPositions}(r_2, s_2, s_3)$ **do**
    for all $r_3' \in \textsc{RealizedPositions}(r_2', s_2, s_3)$ **do**
      add $r_3 \searrow\!\!\swarrow r_3'$ to hp_3

In contrast to Algorithms 8 to 11, the remaining three Algorithms 12 to 14 introduce heap predicates that possibly do not yet exist in the input states.

In Algorithm 12 (AddNewPossibleEquality) we identify two positions leading to different references in an input state, where the other state contains the same reference at these positions. This, again, is done using the information available using MergedTo. As we know that for the two positions the same reference exists in one of the input states, we introduce a $=^?$ heap predicate in s_3.

Algorithm 12: AddNewPossibleEquality

for all (r_1, r_2, r_3) : $\textsc{MergedTo}(r_1, r_2, r_3) \wedge h_1(r_1) \in \textsc{Instances} \cup \textsc{Arrays}$ **do**
  for all (r_2', r_3') : $r_2 \neq r_2' \wedge r_3 \neq r_3' \wedge \textsc{MergedTo}(r_1, r_2', r_3')$ **do**
    add $r_3 =^? r_3'$ to hp_3
  for all (r_1', r_3') : $r_1 \neq r_1' \wedge r_3 \neq r_3' \wedge \textsc{MergedTo}(r_1', r_2, r_3')$ **do**
    add $r_3 =^? r_3'$ to hp_3

To implement the functionality of Definition 1.10(m), we use Algorithm 13 (AddNewJoinsPredicates). First, we identify references which are marked as possibly being equal (lines 2–3), and references reachable using two different positions (lines 4–5). In NeedJoins we first check if for one of the references we have a position which does not exist in s_3 and, if this is the case, if for the two references we have positions such that in s_3 these positions do not lead to the same reference (i.e., the references at the realized prefixes need to be connected using a joins heap predicate), or if the unrealized parts of the positions are different (i.e., the non-tree shape existing in the input state is not represented in s_3 and must be allowed using heap predicates). Then, similar to

MergeCyclicPredicates in Algorithm 8, we identify the references at realized prefixes of the corresponding positions in s_3 using RealizedPositions and add a joins heap predicate.

Algorithm 13: AddNewJoinsPredicates

$\mathit{Check}_1 := \{\}$
for all $r_1 =^? r_1' \in hp_1$ **do**
  $\mathit{Check}_1 := \mathit{Check}_1 \cup \{(r_1, r_1')\}$
for all $r_1 \in$ ReferencesWithMultiplePositions(s_1) **do**
  $\mathit{Check}_1 := \mathit{Check}_1 \cup \{(r_1, r_1)\}$
for all $(r_1, r_1') \in \mathit{Check}_1$ **do**
  if NeedJoins(r_1, r_1', s_1, s_3) **then**
    for all $r_3 \in$ RealizedPositions(r_1, s_1, s_3) **do**
      for all $r_3' \in$ RealizedPositions(r_1', s_1, s_3) **do**
        add $r_3 \curlyvee r_3'$ to hp_3
$\mathit{Check}_2 := \{\}$
for all $r_2 =^? r_2' \in hp_2$ **do**
  $\mathit{Check}_2 := \mathit{Check}_2 \cup \{(r_2, r_2')\}$
for all $r_1 \in$ ReferencesWithMultiplePositions(s_1) **do**
  $\mathit{Check}_2 := \mathit{Check}_2 \cup \{(r_2, r_2)\}$
for all $(r_2, r_2') \in \mathit{Check}_2$ **do**
  if NeedJoins(r_2, r_2', s_2, s_3) **then**
    for all $r_3 \in$ RealizedPositions(r_2, s_2, s_3) **do**
      for all $r_3' \in$ RealizedPositions(r_2', s_2, s_3) **do**
        add $r_3 \curlyvee r_3'$ to hp_3

Finally, in Algorithm 14 (IdentifyNonTreeShapes) we introduce heap predicates for concrete non-tree shapes in the input states that are not represented in s_3, corresponding to Definition 1.10(n). With NonTreeShapes we get α, τ, τ' as in Definition 1.10(n). As the name of NoCommonIntermediateReference indicates, we use this algorithm to check if in the two given positions $\alpha\tau$, $\alpha\tau'$ lead to a common intermediate reference in the given input state when starting in α (cf. Definition 1.9). Again, we make use of RealizedPositions.

Instead of formally showing correctness of the Merge algorithm, which is straightforward in most cases, we recognize that the main difficulty lies in how infinite position sets are handled. Thus, in Section 2.3 we concentrate on the algorithms RealizedPositions, NeedJoins, ReferencesWithMultiplePositions, and NonTreeShapes and explain how these can be implemented.

2.2.1 Example

As promised earlier, we now demonstrate the main aspects of the Merge algorithm by using an example.

Algorithm 14: IDENTIFYNONTREESHAPES

for all $(\alpha, \alpha\tau, \alpha\tau') \in$ NONTREESHAPES(s_1) **do**
  if NOCOMMONINTERMEDIATEREFERENCE$(\alpha\tau, \alpha\tau', s_1)$ **then**
    if $\alpha\tau \notin$ SPOS$(s_3) \vee \alpha\tau' \notin$ SPOS$(s_3) \vee s_3|_{\alpha\tau} \neq s_3|_{\alpha\tau'}$ **then**
      for all $r \in$ REALIZEDPOSITIONS$(s_1|_\alpha, s_1, s_3)$ **do**
        add $r \curlyvee r$ to hp_3
        if $\tau' = \varepsilon$ **then**
          add $r \circlearrowleft_\tau$ to hp_3
for all $(\alpha, \alpha\tau, \alpha\tau') \in$ NONTREESHAPES(s_2) **do**
  if NOCOMMONINTERMEDIATEREFERENCE$(\alpha\tau, \alpha\tau', s_2)$ **then**
    if $\alpha\tau \notin$ SPOS$(s_3) \vee \alpha\tau' \notin$ SPOS$(s_3) \vee s_3|_{\alpha\tau} \neq s_3|_{\alpha\tau'}$ **then**
      for all $r \in$ REALIZEDPOSITIONS$(s_2|_\alpha, s_2, s_3)$ **do**
        add $r \curlyvee r$ to hp_3
        if $\tau' = \varepsilon$ **then**
          add $r \circlearrowleft_\tau$ to hp_3

A

$\langle 23 | \mathsf{one}: r_1, \mathsf{two}: r_9 | r_9 \rangle$

r_1: List(value: i_1, next: r_2)
r_2: List(value: i_1, next: null)
r_9 : List(next : r_{10})
r_{10} : List(self : r_{10})
i_1 : $\{0\}$

B

$\langle 42 | \mathsf{one}: r_3, \mathsf{two}: r_{11} | r_{13} \rangle$

r_3: List(value: i_2, next: r_4)
r_4 : List(next : r_5)
r_5 : List(next : null)
r_{11} : List()
r_{13} : List()
i_2 : $[3, 5]$
$r_{11} \circlearrowleft_{\{\mathsf{next}\}}$
$r_{13} \circlearrowleft_{\{\mathsf{next}\}}$

C

$\langle 42 | \mathsf{one}: r_6, \mathsf{two}: r_{12} | r_{14} \rangle$

r_6: List(value: i_3, next: r_7)
r_7 : List(next : r_8)
r_8 : List()
r_{12} : List()
r_{14} : List()
i_3 : $[0, 5]$
$r_8?$
$r_{12} \circlearrowleft_\emptyset$
$r_{14} \circlearrowleft_\emptyset$
$r_{12} =^? r_{14}$
$r_{12} \curlyvee r_{12}$
$r_{12} \curlyvee r_{14}$
$r_{14} \curlyvee r_{14}$

Figure 2.1.: States illustrating Algorithm 3

Example 2.1 Consider the states shown in Fig. 2.1. We merge states A and B and obtain state C.

We start with MERGEREFERENCES(r_1, r_3). As we did not merge r_1 and r_3 already, the result is obtained by invoking MERGEINSTANCES(r_1, r_3). In MERGEINSTANCES we create a fresh reference r_6 for the result and remember that r_1 and r_3 are merged into r_6. As both instances define the fields value and next, we obtain an object instance where these field are defined. The content of the value field is computed using MERGEREFERENCES(i_1, i_2), while the content of the next field is computed using MERGEREFERENCES(r_2, r_4). The first invocation leads to MERGEPRIMITIVES (which is not shown in this thesis), and we obtain i_3 with the data $[0, 5]$.

Computing MERGEREFERENCES(r_2, r_4) is similar to the previous computation.

However, for r_2 we have the value field information. As this information is not present in r_4, we do not include it in the resulting object instance. The next field again is present in both object instances, and its content is computed by calling MergeReferences(null, r_5).

As one of the arguments is null, we call MergeNull(null, r_5). This call results in an object instance r_8 without field information, which is marked as possibly existing.

Now consider the call MergeReferences(r_9, r_{11}) made for the local variable two. This, again, results in MergeInstances to be invoked. We see that the next field of r_{11} is not represented. Thus, we obtain r_{12} for which no field information is obtained. Note that the cycle represented in A is not traversed during computation of MergeReferences.

Next, we consider the operand stack of the states, leading to MergeReferences(r_9, r_{13}). Based on this we obtain r_{14} in C.

At this point, the loop shown in lines 11–16 of the Merge algorithm is finished. In the next steps, we need to consider the heap predicates.

We start with MergeCyclicPredicates. The only cyclic heap predicate exists for r_{11} and r_{13} in B. The realized positions for r_{11} in C only lead to r_{12}, while for r_{13} we get r_{14}. Thus, we add $r_{12}\circlearrowleft_{\{\mathsf{next}\}}$ and $r_{14}\circlearrowleft_{\{\mathsf{next}\}}$ to C. Note that the added heap predicates do not allow cycles which only traverse the self field, as it is the case for r_{10}.

The algorithms MergePossibleExistence, MergePossibleEquality, and MergeJoinsPredicates copy over existing heap predicates, similar to MergeCyclicPredicates. However, as no such heap predicates exist in A or B, these algorithms are not considered in this example.

Next, we consider AddNewPossibleEquality. The idea of this algorithm is to add the $=^?$ heap predicate to two references which correspond to the same reference in one of the input states. In our example, this is the case for r_{12} and r_{14}, which both result out of r_9 in A. The algorithm detects this by investigating the information we add every time we ran MergeReferences. Here, we added the information $(r_9, r_{11}) \mapsto r_{12}$ and $(r_9, r_{13}) \mapsto r_{14}$. Thus, we add $r_{12} =^? r_{14}$.

Next, we look at AddNewJoinsPredicates. Here, we see that r_9 and r_{10} can be reached by more than one position in A. Thus, we add (r_9, r_9) and (r_{10}, r_{10}) to $Check_1$. For r_9 we see that in C the positions do not lead to the same reference. However, as all positions also exist in C, we do not need to add a joins heap predicate. This is different in the case of r_{10}. Here, we see that we need to add $r_{12} \searrow\!\swarrow r_{12}$, $r_{12} \searrow\!\swarrow r_{14}$, and $r_{14} \searrow\!\swarrow r_{14}$.

Finally, in IdentifyNonTreeShapes we see that we have a cycle in A. This cycle (leading from r_{10} back to r_{10} using the field self) causes us to add heap predicates to r_{12} and r_{14} in C. We already have $r_{12} \searrow\!\swarrow r_{12}$ and $r_{14} \searrow\!\swarrow r_{14}$. Furthermore, we already have $\circlearrowleft_{\{\mathsf{next}\}}$ for these two references. However, we need to add $\circlearrowleft_{\{\mathsf{self}\}}$, which means that we intersect the field sets and obtain $\circlearrowleft_\emptyset$ for both r_{12} and r_{14}.

2.3. State Positions

As for states containing cyclic data structures the set of state positions is infinite, actually implementing the aforementioned algorithms using the concepts described so far is non-trivial. However, even in the presence of cycles the set of references in any state is finite.

2.3.1. RealizedPositions

To compute REALIZEDPOSITIONS, we need to consider a finite subset of state positions. In this subset we limit the loop traversals represented in each contained position. First we consider state positions without any loop traversal. In Definition 2.2 for any position π we define $\widehat{\pi}$ as its cycle-free variant. Since we use an inductive definition, it is clear how to compute $\widehat{\pi}$ for any position π.

Definition 2.2 ($\widehat{\boldsymbol{\pi}}$) Let $\pi \in \mathrm{SPOS}(s)$. If $|\pi| = 1$, then $\widehat{\pi} := \pi$. Otherwise, if we have $\tau \neq \varepsilon$ with $s|_{\pi_1} = s|_{\pi_1\tau}$ and $\pi = \pi_1\tau\pi_2$ (i.e., τ traverses a cycle in s), then $\widehat{\pi} := \widehat{\pi_1\pi_2}$. If no such τ exists, then $\widehat{\pi} := \pi$.

In addition to cycle-free positions we also need to consider positions that contain at most a single cycle traversal. If a position π has this property, we indicate this as $[^{\leq 1c}\pi]$. A position π traverses *exactly* one cycle in s if

- $\pi = \pi_1\tau\pi_2$ with $\tau \neq \varepsilon$ and $s|_{\pi_1\tau} = s|_{\pi_1}$ (i.e., it contains a cycle τ), and
- $\hat{\pi} = \pi_1\pi_2$ (i.e., τ is the only part of the position containing a cycle), and
- the references at $s|_{\pi_1\rho}$ for $\varepsilon \trianglelefteq \rho \triangleleft \tau$ are different (i.e., τ only is a single cycle traversal without any subcycles).

Definition 2.3 ($[^{\leq 1c}\boldsymbol{\pi}]$) Let $\pi \in \mathrm{SPOS}(s)$. If $\pi = \widehat{\pi}$, then we have $[^{\leq 1c}\pi]$. Otherwise, if π cannot be decomposed as $\pi = \pi_1\tau\pi_2$ where $\widehat{\pi} = \pi_1\pi_2$ (i.e., $s|_{\pi_1} = s|_{\pi_1\tau}$), then we do not have $[^{\leq 1c}\pi]$. Finally, let $\pi = \pi_1\tau\pi_2$ with $s|_{\pi_1} = s|_{\pi_1\tau}$ and $\tau \neq \varepsilon$. Then $[^{\leq 1c}\pi]$ iff $|\{s|_{\pi_1\rho} \mid \varepsilon \trianglelefteq \rho \triangleleft \tau\}| = |\tau|$.

For RealizedPositions(r, s, s') we need to compute a set $\Psi := \{s'|_{\overline{\pi}} \mid \pi \in \Pi\}$ where $\Pi := \{\pi \mid s|_\pi = r\}$. In the case of a cyclic data structure in s, Π is infinite. Thus, the set notation of Ψ may not be helpful to actually compute the (finite) set Ψ in finite time. Instead, we show how to compute a finite set $\Omega \subseteq \Pi$ so that $\Psi = \{s'|_{\overline{\pi}} \mid \pi \in \Omega\}$.

The main idea is to limit the infinite set Π to the finite subset that only contains positions where each contained cycle is traversed at most once.

Definition 2.4 (Ω) Let r be a reference in a state s. Let $\Pi = \{\pi \mid s|_\pi = r\}$. Then $\Omega := \{\pi \mid \pi \in \Pi \wedge [^{\leq 1c}\pi]\}$.

Now we show that indeed $\Psi = \{s'|_{\overline{\pi}} \mid \pi \in \Omega\}$. For this we show that for all $\pi \in \Pi \setminus \Omega$ there is a $\pi' \in \Omega$ with $s'|_{\overline{\pi'}} = s'|_{\overline{\pi}}$. In other words, even if π is not contained in Ω, we have another reference π' that gives us the same reference in s'.

The main idea in this proof is to not consider cycle traversals contained in π if the corresponding cycle also can be traversed in s'. Thus, by considering a position that contains at most a single cycle traversal, using π' we can reach the same end reference as π does in s'.

Lemma 2.2 Let s' be a state. Let r, s, Π, Ω as in Definition 2.4.
Then $\{s'|_{\overline{\pi}} \mid \pi \in \Pi\} = \{s'|_{\overline{\pi}} \mid \pi \in \Omega\}$.

Proof. As $\Omega \subseteq \Pi$ we only need to show that for each $\pi \in \Pi \setminus \Omega$ there is $\pi' \in \Omega$ with $s'|_{\overline{\pi'}} = s'|_{\overline{\pi}}$.

Thus, let $\pi \in \Pi \setminus \Omega$. We know $\widehat{\pi} \in \Omega$. If $s'|_{\overline{\widehat{\pi}}} = s'|_{\overline{\pi}}$, the claim directly follows. Thus, assume $s'|_{\overline{\widehat{\pi}}} \neq s'|_{\overline{\pi}}$. Then we know π can be split as $\pi = \pi_0 \alpha_1 \pi_1 \cdots \alpha_n \pi_n$ where $n > 0$, $\widehat{\pi} = \pi_0 \cdots \pi_n$, $\alpha_i \neq \varepsilon$, and $s|_{\pi_0 \cdots \pi_i} = s|_{\pi_0 \cdots \pi_i \alpha_{i+1}}$ for all $1 \leq i < n$.

With $s'|_{\overline{\widehat{\pi}}} \neq s'|_{\overline{\pi}}$ we also know that there is $1 \leq i < n$ with $\pi_0 \cdots \pi_i \trianglelefteq \overline{\pi_0 \cdots \pi_i \alpha_{i+1}}_{s'} \triangleleft \pi_0 \cdots \pi_i \alpha_{i+1}$ (i.e., part of the cycle is not realized in s') or $s'|_{\pi_0 \cdots \pi_i \alpha_{i+1}} \neq s'|_{\pi_0 \cdots \pi_i}$ (i.e., the cycle does not end in the reference at the start of the cycle).

So far, we did not limit the shape of the cycle α_{i+1}, thus it may contain repetitions of a single cycle traversal, or subcycles. Thus, by removing subcycles from α_{i+1} we obtain α'_{i+1} with $[^{\leq 1c}\alpha'_{i+1}]$ and $\pi_0 \cdots \pi_i \trianglelefteq \overline{\pi_0 \cdots \pi_i \alpha'_{i+1}}_{s'} \triangleleft \pi_0 \cdots \pi_i \alpha'_{i+1}$ or $s'|_{\pi_0 \cdots \pi_i \alpha'_{i+1}} \neq s'|_{\pi_0 \cdots \pi_{i+1}}$. Hence, we also have $\pi_0 \cdots \pi_i \alpha'_{i+1} \pi_{i+1} \cdots \pi_n \in \Omega$ with $s'|_{\overline{\pi_0 \cdots \pi_i \alpha'_{i+1} \pi_{i+1} \cdots \pi_n}} = s'|_{\overline{\pi}}$. □

As we now have shown that the finite set Ω suffices for the task of computing RealizedPositions, we finally show that Ω can be computed using ComputeOmega

(Algorithm 15). In the first part (lines 1–16) we traverse the state using all possible positions. If we run through a cycle (line 6), we stop traversing the state with suffixes of the current position. Furthermore, we remember the cycle in *Continuations*. For each reached position π', which by construction is cycle-free, we store all $\tau \neq \varepsilon$ with $s|_{\pi'} = s|_{\pi'\tau}$ where $\pi'\tau$ contains exactly one cycle. In *Positions* we collect all cycle-free positions of the state.

The second part of the algorithm, lines 18–24, uses the information computed in the first part. For each cycle-free position π, we mark π as part of the result. Furthermore, we also add positions based on π where we added a single cycle traversal. For this, we consider all prefixes of π and, for each prefix where a cycle is known, we add the position resulting out of adding a cycle to the result set.

Using Algorithm 15 it is trivial to implement RealizedPositions such that we have RealizedPositions$(r, s, s') = \{s'|_{\overline{\pi}} \mid \pi \in \Omega\} = \{s'|_{\overline{\pi}} \mid s|_{\pi} = r\}$.

Algorithm 15: ComputeOmega

Input: $s \in$ States, $r \in$ References
Output: Ω as in Definition 2.4

1: *Positions* $:= \emptyset$
2: *Continuations* $:= \emptyset$
3: *Todo* $:= \{\pi \in \mathrm{SPos}(s) \mid |\pi| = 1\}$
4: **while** $\exists \pi : \pi \in$ *Todo* **do**
5:    *Todo* $:=$ *Todo* $\setminus \{\pi\}$
6:    **if** $\exists \pi', \tau \neq \varepsilon$ with $\pi = \pi'\tau \wedge s|_{\pi'} = s|_{\pi'\tau}$ **then**
7:       add τ to *Continuations*(π')
8:    **else**
9:       add π to *Positions*
10:       **if** $h(\pi) = f \in$ Instances **then**
11:          **for all** $v \in \mathrm{dom}(f)$ **do**
12:             *Todo* $:=$ *Todo* $\cup \{\pi v\}$
13:       **else if** $h(\pi) = (i_l, f) \in$ Arrays **then**
14:          *Todo* $:=$ *Todo* $\cup \{\pi \mathsf{len}\}$
15:          **for all** $i \in \mathrm{dom}(f)$ **do**
16:             *Todo* $:=$ *Todo* $\cup \{\pi i\}$
17:
18: *Result* $:= \emptyset$
19: **for all** $\pi \in$ *Positions* **do**
20:    add π to *Result*
21:    **for all** $\pi_1\pi_2 = \pi \wedge \pi_1 \neq \varepsilon$ **do**
22:       **for all** $\tau \in$ *Continuations*(π_1) **do**
23:          add $\pi_1\tau\pi_2$ to *Result*
24: **return** *Result*

2.3.2. NeedJoins

In AddNewJoinsPredicates we use NeedJoins to find out whether a joins heap predicate needs to be added. By using the positions computed in ComputeOmega, this can easily be determined. For the given two references we first compute two sets of positions using ComputeOmega. Then for each combination of positions contained in the two sets we check the requirements as stated in Definition 1.10(m).

2.3.3. ReferencesWithMultiplePositions

Here, we can also use ComputeOmega by computing the positions for each (non-primitive) reference known in the state. Then we return the references for which ComputeOmega returns more than one position.

2.3.4. NonTreeShapes

The algorithm NonTreeShapes should return all positions $\alpha, \alpha\tau, \alpha\tau'$ such that $s|_{\alpha\tau} = s|_{\alpha\tau'}$ where $\tau \neq \varepsilon$. Again, there may be infinitely many such (triples of) positions. However, again it is safe to ignore positions α containing cycles that also are realized in s_3. Thus, by considering only the positions as computed by ReferencesWithMultiplePositions and then computing α, τ, τ' based on these, we can easily implement NonTreeShapes.

2.4. Instance Check

In Section 2.2 we presented an algorithm enabling us to merge two states. However, for the graph construction we not only need to merge states, but we also need to check if $s_1 \sqsubseteq s_2$ holds. Instead of using another algorithm for this task, we re-use the Merge algorithm and add little modifications that enable us to also compute $s_1 \sqsubseteq s_2$ using this already existing algorithm.

When determining if $s_1 \sqsubseteq s_2$ holds, we compute Merge(s_1, s_2). Whenever we add information from s_1 to s_3 , we check if all of this information is also represented in s_2. For example, in line 3 of Algorithm 9 we add $r_3?$ to s_3 because we have the heap predicate $r_1?$ for a reference r_1 in s_1. If we do not have $r_2?$, we know that either

- r_2 is null and, thus, in s_2 we do not consider the possibility that r_2 exists, or
- r_2 is known to be existing and we do not consider the possibility that r_2 is null.

In both cases, we have $s_1 \not\sqsubseteq s_2$. Similarly, if in MergeArrays we have the case that $h_1(r_1) \in$ Instances and $h_2(r_2) \in$ Arrays (lines 4–5 in Algorithm 6), we know that

$s_1 \not\sqsubseteq s_2$ as in s_1 the referenced instance may be a (non-array) object, while it is known to be an array in s_2.

Thus, as soon as a conflict as indicated above is found in the merge process, we know $s_1 \not\sqsubseteq s_2$. Only if merging is possible without finding information in s_1 that is not represented in s_2, we know that $s_1 \sqsubseteq s_2$ holds.

As the shape of s_1 and s_2 may be different, additional (simple) checks need to be implemented.

3. Recursion

In this chapter we extend the technique presented in Chapter 1 so that also recursive programs may be analyzed. The reason for the limitation in the previously presented approach is that in recursive programs the call stack may grow without bounds. As in our setting we usually provide method arguments as abstract values, we may not only obtain call stacks of unbounded height for non-terminating methods, but also for methods where the execution depends on the input values.

Example 3.1 When computing the factorial function n! using the method factorial, the height of the call stack may be n. If n is not a concrete value, but instead an abstract value like $[0, \infty) \in$ Integers, we cannot provide a finite upper bound to the call stack height, even though factorial always terminates.

```
public int factorial(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

For the corresponding problem in non-recursive programs (i.e., loops that may be non-terminating) we abstract the data contained in the states to obtain a finite representation of all possible computations. However, abstraction of the call stack is not as simple.

Call Stack Abstraction

The frames in the call stack contain information about local variables and the operand stack. Thus, in order to abstract the call stack, we need a way to abstract this data. While for non-recursive programs the number of state positions of length one (i.e., local variables, operand stack entries, static fields, and the exception reference) is bounded by some finite number, for recursive methods the number of such positions in a state may grow without bounds. In Example 3.1, for each frame on the call stack a different value of n is stored.

To work around this problem, in our analysis we do not retain information for each frame of the call stack, as in the case of recursion there may be an unbounded number of stack frames corresponding to the same recursive method. Instead, we only represent at most one stack frame corresponding to a recursive method in each state and disregard information about the other stack frames. In the case of Example 3.1 this also means that for a concrete state containing several stack frames for factorial, in the constructed Symbolic Execution Graph we only have abstract states representing the topmost of these stack frames.

As the data of the lower stack frames corresponding to calls to recursive methods is not part of our state representation, suitable abstract values must be inferred when continuing evaluation after a recursive method returns. In Example 3.1, as only information about the topmost stack frame of the factorial recursion is represented in our abstract state, when returning from factorial invoked from a recursive call in line 5, we not only need the returned value to continue evaluation, but also need the value of n stored in the frame below the one we are returning from, as it is used in a multiplication.

For primitive values, as in Example 3.1, this problem can easily be solved by just assuming that parts of the heap which are not explicitly represented contain $(-\infty, \infty) \in$ Integers or $\perp \in$ Floats. However, in the case of object instances and arrays a similar idea directly leads to states containing very little information: each object instance may be null, it may be of any type, it may be cyclic, and it may share with any other object instance. For most subsequent analyses this huge loss of information is not acceptable, e.g., proving termination usually is problematic if possibly cyclic objects are part of the computation.

Example 3.2 The method length recursively computes the length of the List object provided as this. Assuming that the list is acyclic, this algorithm terminates.

In test we compute the length *twice*. If after returning from the first call we do not have the information that list is acyclic (if we do not have any information about the lower stack frames when returning from the last frame of length) we cannot show termination of the second invocation of length.

```
1  public class List {
2    List next;
3
4    public int length() {
5      if (next == null) {
6        return 1;
7      } else {
8        return 1 + next.length();
9      }
10   }
11
12   public static void test(List list) {
13     list.length();
14     list.length();
15   }
16 }
```

One idea to solve this issue is to combine the information of the return state with information contained in the state of the call site. The lower stack frames are not represented in the return state, but this information is part of the state of the call site. In a sense, the call site defines the *context* of the invocation and determines where evaluation continues.

When returning from a method we create states corresponding to each possible call site so that all possible execution paths are represented in the resulting Symbolic Execution Graph. Furthermore, we can re-use the information available in the lower stack frames, and thus avoid the aforementioned problem. We call this process *context concretization*.

Using context concretization, for Example 3.2 the state of the call site in line 13 indicates that list is acyclic. Thus, we can retain this information when returning and continuing analysis with the call in line 14.

Side Effects

If in Example 3.2 at the end of the recursion (at the end of the list, in line 6) we insert the additional code next = this, we modify the list to be cyclic and, thus, must not show termination of any following invocation of length on the same list. This also means that the invocation in line 14 of test does not terminate for the algorithm modified as explained above. Thus, when returning from the first invocation of length in line 13, we may not assume that list is acyclic, although this was the case *before* the invocation. However, the state of the call site corresponding to the invocation in line 13 contains the information that the list is acyclic. Thus, we may not re-use this information when returning from this invocation and constructing the context concretization.

In Chapter 1, we add heap predicates to mimic the effects of write accesses to references for which the connections to the modified parts of the heap are not represented explicitly. For the example above, Definition 1.43 assures that a $\circlearrowleft$ heap predicate is introduced where necessary. However, in the recursive example we need to add the heap predicate for a reference that is not represented in the state where we evaluate the write access. Thus, even in the presence of side-effects, the information in the states of the call sites is not updated and, thus, may contain invalid information which we may not re-use after the invoked method returns.

In order to be able to remember that a write-access may have changed information that is part of the state of a call site, we extend the definition of states and introduce the concept of *input arguments* as another component. The details of this component will be defined in the course of this chapter. For now, it suffices to understand that the contained information helps us to propagate changes (side-effects) during execution of some method so that, when returning and constructing the context concretization, we can disregard the outdated, invalidated information in the state of the call state. As such, the concept of using additional input arguments is similar to the idea of using *shadow variables* which, for example, are also used in COSTA [AAC+08].

Furthermore, if we know that some data was *not* changed, we may be able to combine the information of the return state and the state of the call site, possibly creating more precise information in the context concretization.

Idea of Graph Construction

We now explain the idea of how the graph construction as explained in Chapter 1 is adapted. For this, we illustrate all main concepts presented in this chapter using a simple example. In Fig. 3.1 we develop an example Symbolic Execution Graph for factorial as shown in Example 3.1. First, consider Fig. 3.1a. Analysis starts with the start state s_0. After a refinement corresponding to `if (n <= 1)` in line 2 of the program, the left branch leads to a program end, shown as s_ε with an empty call stack. The predecessor state s only contains a single stack frame and corresponds to return 1 in line 3. We call states like s *return states*. On the right we have the recursive call to factorial, which corresponds to line 5 in the code. Let $\hat{s}$ be the state created as the result of evaluating the call, such that $\hat{s}$ contains two stack frames for the factorial method – one corresponding to the invocation starting in s_0, and one on top corresponding to the recursive call. We call states like $\hat{s}$ *invoking states*. As explained earlier, we do not want to have states containing more than one stack frame corresponding to any recursive method, as this might lead to a call stack of unbounded height. Indeed, continuing analysis of $\hat{s}$ using only the concepts of

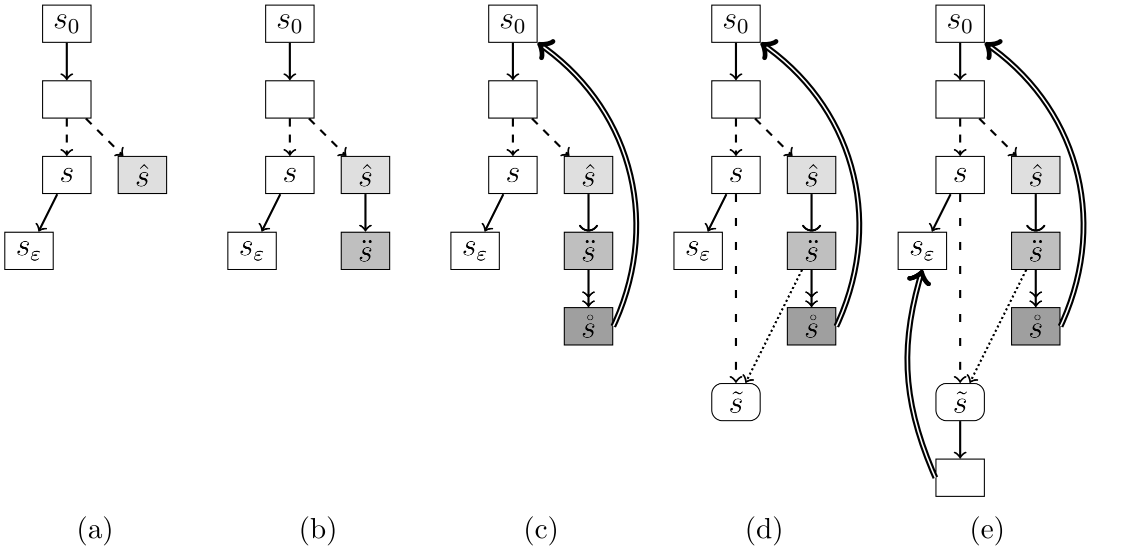


Figure 3.1.: Constructing a Symbolic Execution Graph for a recursive program

Chapter 1 would lead to an infinite graph, assuming an upper bound of ∞ for num.

Instead, starting with $\hat{s}$ we make use of new concepts. In a first step, shown in Fig. 3.1b, we create a *call state* $\ddot{s}$ which is a copy of $\hat{s}$, but in addition the top stack frame (which was just created for the recursive call) contains *input arguments*. We connect $\hat{s}$ to $\ddot{s}$ using an *input arguments creation edge*.

Then, shown in Fig. 3.1c, based on $\ddot{s}$ we create a state only containing the topmost stack frame, shown as $\mathring{s}$. We connect $\ddot{s}$ to $\mathring{s}$ using a *call edge*. This abstraction of $\ddot{s}$ still contains the input arguments created for $\ddot{s}$ and we call such states *call stack abstraction states*. If now also those input arguments are represented in s_0 and we have $\mathring{s} \sqsubseteq s_0$, we connect $\mathring{s}$ to s_0 using an instance edge.

The graph Fig. 3.1c does not contain states corresponding to code following a return from a recursive call. In the example of factorial, the multiplication in line 5 is not shown. Thus, in Fig. 3.1d we extend the graph by adding the *context concretization* $\tilde{s}$ of s and $\ddot{s}$. This state $\tilde{s}$ corresponds to the code following the recursive call in $\ddot{s}$ where the invoked method returns as described in s. We add a *context concretization edge* from s to $\tilde{s}$. With this information we then continue the graph construction, starting with dropping the top stack frame according to the RETURN opcode inherited from s. This leads to another program end.

As we created another return state corresponding to line 5 in the program, we now need to perform context concretization with this return state and $\ddot{s}$. While this is not shown in Fig. 3.1, we already introduced all concepts needed to construct a Symbolic Execution Graph for recursive programs.

Structure

In Section 3.1 we present related work. Then, in Section 3.2 we introduce the adaptations to abstract states as defined in Chapter 1 and also define states with a special role in the process of context concretization, for example call states. In Section 3.3 we then explain how context concretization is performed. A large part (indeed, the largest part of this chapter) is devoted to showing stability of $\sqsubseteq$ under context concretization in Section 3.4. This result forms the basis of most of the upcoming proofs. Then, in Section 3.5 we show how the concept of context concretization is used to construct Symbolic Execution Graphs, making use of several new types of edges. In Section 3.6 we discuss a problem that causes the created graphs to be infinite. Finally, in Section 3.7 we conclude and give an outlook on possible extensions.

3.1. Related Work

There are at least two main approaches for analyzing recursive programs. One such approach is to compute *summaries* which describe the effects of the invoked methods. Then, whenever a method is invoked, the effects described by the corresponding summary are applied instead of continuing analysis of the invoked method. Of course, in the case of recursive methods computing such summaries is not easy. The approach of summary computation is discussed, for example, in [CPR09, CDOY09, RHS95, JLRS04].

The tools Julia [SMP10] and COSTA [AAC⁺08], which were introduced previously, also are able to analyze recursive Java Bytecode programs. Here, the heap is abstracted using integers and, as such, the effects of recursive method invocations can be computed comparatively easy.

Another approach is to *inline* function calls, possibly resulting in states with a call stack of unbounded height. Here, an abstraction of the call stack is performed. In [RC11] the authors present how the call stack can be abstracted using techniques similar to those used to abstract shapes on the heap.

In [BOG11] we already presented the key concepts of the technique presented in this chapter. However, we did not allow the usage of heap predicates (then named *annotations*) in the states.

3.2. States

We need to add *input arguments* to our state definition, so that side effects can be detected and made visible when performing context concretization.

Definition 3.2 (Input Arguments) Let $\textsc{InputArguments}$ be a new component used in the upcoming state definition. For that we define

$$\textsc{InputArguments} = 2^{\textsc{References} \times (\textsc{States} \mapsto \textsc{References}) \times \mathbb{B}}$$

Here, the first component describes the reference used in the state for the input argument. The second component is a function that, given a state, gives a reference in that state which is represented by the input argument. In the third component we encode if the input argument may have been changed, or is left unchanged.

We will usually write $(\lambda, \gamma, \checkmark/\lightning)$ to represent a single input argument where $\checkmark/\lightning$ indicates that we know if the input argument is left unchanged, or not. If we know that an input argument is left unchanged, we write $(\lambda, \gamma, \checkmark)$. For possibly changed input arguments we write $(\lambda, \gamma, \lightning)$.

Using Definition 3.2 we now re-define states such that stack frames may contain input arguments.

Definition 3.3 (Abstract States with Input Arguments) We extend Definition 1.1 by adding a new component $\textsc{InputArguments}$ to each frame of the call stack.

$$\begin{aligned}\textsc{CallStack} :=& (\textsc{ProgramPositions} \times \textsc{LocalVariables} \\ & \times \textsc{OperandStack} \times \textsc{InputArguments})^*\end{aligned}$$

From now on, for states we will use the standard notation

$$s = (\langle fr_0, \ldots, fr_n \rangle, h, t, hp, sf, e, ic, sr) \text{ with } fr_i = (pp_i, lv_i, os_i, ia_i)$$

Furthermore, let $|s|$ be the height of the call stack of s. A state s is concrete only if all $\textsc{InputArguments}$ components are empty.

With input arguments in a state, we also have state positions for the corresponding references.

Definition 3.4 (State Positions) Let $s = (\langle fr_0, \ldots, fr_n \rangle, h, t, hp, sf, e, ic, sr)$ be a state where each stack frame fr_i has the form (pp_i, lv_i, os_i, ia_i). Then $\textsc{SPos}(s)$ is the smallest set containing all of the following sequences π (where we just added the first entry in addition to those of Definition 1.5:

- $\pi = \text{IA}_{i,\gamma}$ where $(\lambda, \gamma, \checkmark/\lightning) \in ia_i$ for some $0 \leq i \leq n$. Then $s|_\pi$ is λ.
- $\pi = \text{LV}_{i,j}$ where $0 \leq i \leq n, lv_i = r_{i,0}, \ldots, r_{i,m_i}, 0 \leq j \leq m_i$. Then $s|_\pi$ is $r_{i,j}$.
- $\pi = \text{OS}_{i,j}$ where $0 \leq i \leq n, os_i = r_{i,0}, \ldots, r_{i,m_i}, 0 \leq j \leq m_i$. Then $s|_\pi$ is $r_{i,j}$.
- $\pi = \text{SF}_v$ where $sf(v) = r$. Then $s|_\pi$ is r.
- $\pi = \text{EXC}$ where $e = r \neq \bot$. Then $s|_\pi$ is r.
- $\pi = \pi'\, v$ for some $v \in \textsc{FieldIDs}$ and some $\pi' \in \text{SPos}(s)$ where $h(s|_{\pi'}) = f \in \textsc{Instances}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.
- $\pi = \pi'\, i$ for some $i \in \mathbb{N}$ and some $\pi' \in \text{SPos}(s)$ where $h(s|_{\pi'}) = (i_l, f) \in \textsc{Arrays}$ and where $f(i)$ is defined. Then $s|_\pi$ is $f(i)$.
- $\pi = \pi'\, \mathsf{len}$ for some $\pi' \in \text{SPos}(s)$ where $h(s|_\pi) = (i_l, f) \in \textsc{Arrays}$. Then $s|_\pi$ is i_l.

In the explanations so far, we introduced several new kinds of states, for example call states. We will now formally define these states.

An *invoking state* contains information about the invoked method (in the topmost stack frame) and in the remainder of the stack the context of this invocation, including the used arguments, is available. In addition, the topmost stack frame does not contain any input argument. Invoking states are created using evaluation and abstraction as defined in Chapter 1.

Definition 3.5 (Invoking State) We call s an *invoking state* if the topmost stack frame does not contain any input argument, i.e., $ia_0 = \emptyset$. Furthermore we demand that

- in the graph there is a state s' with $s' \xrightarrow{\textsc{Eval}} s$ and $|s| = |s'| + 1$, or
- in the graph there is a state s' with $s' \xrightarrow{\textsc{Instance}} s$ and s' is an invoking state.

For each invoking state s we have $|s| > 1$. Also note that in Java Bytecode the exception reference may not be set when a method is invoked, i.e., for an invoking state s we always have $s|_{\text{EXC}} = \bot$. Furthermore, in the topmost stack frame the operand stack is empty, i.e., there is no position $\pi \in \text{SPos}(s)$ with $\pi \trianglerighteq \text{OS}_{0,j}$ for any j.

Based on an invoking state, we create a corresponding *call state*. A call state contains the input arguments we need to detect side effects visible to references in the lower stack frames, which are not represented in the analysis following the call state. In order to be able to detect all side effects, we need to have input arguments for all arguments of the invoked method. Furthermore, as references stored in static fields may also be changed by the invoked method, we also provide input arguments for these. These ideas are represented in Definition 3.6(ii).

Additionally, certain predecessors and all successors of these changeable references also need to be provided using input arguments. As an example, we might have two arguments provided in local variables x and y with a common predecessor reference. If we now connect x and y using a write access like x.f = y, this predecessor now represents a non-tree shape on the heap. To have the necessary information, we demand input arguments for predecessors in Definition 3.6(iv).

For technical reasons, we add input arguments according to Definition 3.6(v).

With Definition 3.6(vi) we limit that input arguments may only be created for references as described in Definition 3.6(ii–v).

Note that using abstraction of the preceding invoking state, it is possible to decrease the number of necessary input arguments. For example, if one decides to not represent successors of arguments provided to the method explicitly (i.e., one ensures that $\mathrm{dom}(f) = \emptyset$ for the corresponding object instances), this may cause less input arguments to be created according to Definition 3.6(ii).

Analogously, if one decides not to allow explicit connections from some predecessor reference *to* a reference provided as an argument, possibly less input arguments need to be created according to Definition 3.6(iv).

Definition 3.6 (Call State) In the course of this chapter, call states will be denoted as $\ddot{s} = (\langle \ddot{fr}_0, \ldots, \ddot{fr}_m \rangle, \ddot{h}, \ddot{t}, \ddot{hp}, \ddot{sf}, \bot, \ddot{ic}, \bot)$ with $\ddot{fr}_i = (\ddot{pp}_i, \ddot{lv}_i, \ddot{os}_i, \ddot{ia}_i)$. We call $\ddot{s}$ a *call state* if

(i) in the graph there is an invoking state s with $s \xrightarrow{\text{IA CREATION}} \ddot{s}$

(ii) an input argument exists for every reference reachable from the invoked method: $\forall \pi \in \{\mathrm{SF}_v, \mathrm{LV}_{0,j}\} : \ddot{s}|_\pi \rightsquigarrow \ddot{\lambda} \to \exists(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$

(iii) for each $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\mathrm{IA}_{0,\ddot{\gamma}}\, v \in \mathrm{SPos}(\ddot{s})$, we also have $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{\lambda}' = \ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}\, v}$.

(iv) an input argument exists for all predecessors of reachable references: $\forall \pi\ \forall \pi' : \ddot{\lambda} \rightsquigarrow \ddot{s}|_{\pi'} \wedge \ddot{s}|_\pi \rightsquigarrow \ddot{s}|_{\pi'} \wedge \pi \in \{\mathrm{SF}_v, \mathrm{LV}_{0,j}\} \to \exists(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$

(v) for each $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{\lambda} =^? \ddot{\lambda}'$ we have $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$.

(vi) for each $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ we have $\ddot{s}|_\pi \rightsquigarrow \ddot{\lambda}$, or $\ddot{\lambda} \rightsquigarrow \ddot{s}|_{\pi'} \wedge \ddot{s}|_\pi \rightsquigarrow \ddot{s}|_{\pi'}$, or $\ddot{\lambda} =^? \ddot{\lambda}' \rightsquigarrow \ddot{s}|_{\pi'} \wedge \ddot{s}|_\pi \rightsquigarrow \ddot{s}|_{\pi'}$ for any $\pi \in \{\mathrm{SF}_v, \mathrm{LV}_{0,j}\}$ and any $\pi', \ddot{\lambda}'$.

(vii) the reference of each input argument is referenced by the corresponding mapping: $\forall(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0 : \ddot{\gamma}(\ddot{s}) = \ddot{\lambda}$ (i.e., $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = \ddot{\lambda}$)

We now demonstrate Definition 3.6 using an example.

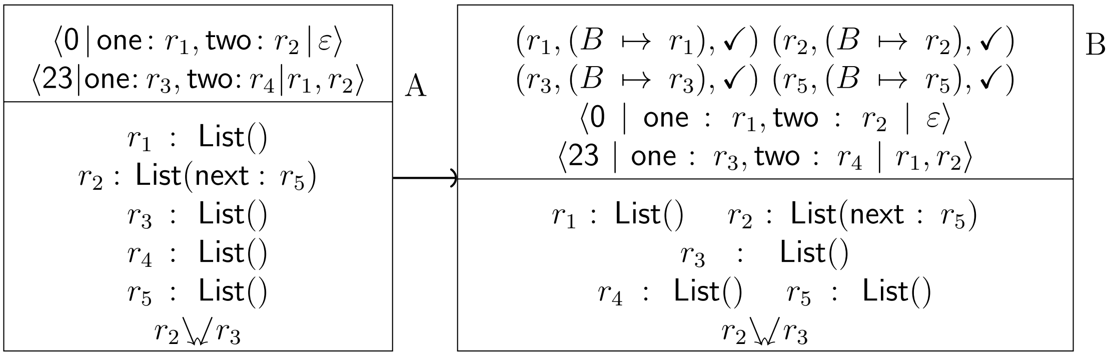


Figure 3.7.: States illustrating Definition 3.6

Example 3.3 Consider states A and B shown in Fig. 3.7. Here, state A is an invoking state, connecting to the call state B. The input arguments of B are shown above the corresponding stack frame.

First, we see that according to Definition 3.6(ii) we need to have input arguments for r_1 and r_2, as these references are available in local variables of the topmost stack frame. Thus, the invoked method may access the referenced data through these local variables.

Then, as r_5 is a successor of r_2, we also need to have an input argument for r_5 according to Definition 3.6(iii).

With Definition 3.6(iv) we also need to have an input argument for r_3, as r_3 is a possible predecessor of r_2, for which we have an input argument. This way, if r_2 is modified, we also know that the write access may have influenced the predecessor r_3.

The condition of Definition 3.6(v) is not met, thus we do not need to add additional input arguments.

The limitation stated in Definition 3.6(v) states that for every created input argument certain conditions need to be met. This is the case in our example, as we only introduced input arguments which are necessary according to other parts of the definition.

Finally, according to Definition 3.6(vii) we also need to make sure that the mapping function references the reference used to represent the input argument. In this example we just re-used the reference names we already had in state A.

A call state $\ddot{s}$ always represents *older* information when compared to a state s which results out of the invocation started in $\ddot{s}$. When computing the context concretization of $\ddot{s}$ and s, we only are interested in changes applied during the evaluation leading from $\ddot{s}$ to s, as these represent side effects which possibly are visible when returning from the method and continuing analysis following the invocation in $\ddot{s}$. In other words, when performing context concretization with $\ddot{s}$ and s, the information in $\ddot{s}$ either is *still valid* or *outdated*, but it never is *more recent* than the information in s. Thus, in the next definition we say that a reference is marked as changed not only if a corresponding input argument with this information exists, but we also demand that the reference is *not* contained in a call state.

Definition 3.8 (Changed References $r\lightning$, Unchanged References $r\checkmark$) Let r be a reference in a state s. We define that $r\lightning$ holds iff s is not a call-state and there is an input argument $(\lambda, \gamma, \lightning) \in ia_n$ with $s|_{\mathrm{IA}n,\gamma} \rightsquigarrow r$. If for r we do not have $r\lightning$, we have $r\checkmark$.

Now we define the remaining states which we need to address when constructing Symbolic Execution Graphs for recursive methods. A *program end* is a state in which the program ends and no further evaluation is possible.

Definition 3.9 (Program End) A state s is a program end if $|s| = 0$, i.e., it does not contain any stack frame.

When returning from a recursive call, the Symbolic Execution Graph contains a corresponding program end. However, as the call stack of a program end is empty, we do not have any information about, for example, a returned reference. Thus, for each program end we define the corresponding *return state* which, when evaluated, leads to a program end.

Definition 3.10 (Return State) A state s is a return state if there is a program end s' with $s \xrightarrow{\text{EVAL}} s'$. Note that we must have $|s| = 1$.

Note that return states contain a RETURN opcode in the topmost and only stack frame, or an exception is thrown but not caught in the current method.

Finally, we also consider *call stack abstraction states*, e.g. $\mathring{s}$ in Fig. 3.1c. In a call stack abstraction state we only represent the topmost stack frame of the corresponding call state.

Definition 3.11 (Call Stack Abstraction State)
Let $\ddot{s} = (\langle \ddot{fr}_0, \ldots, \ddot{fr}_m \rangle, \ddot{h}, \ddot{t}, \ddot{hp}, \ddot{sf}, \bot, \ddot{ic}, \bot)$ be a call state. Then we define that

$$s = (\langle \ddot{fr}_0 \rangle, \ddot{h}, \ddot{t}, \ddot{hp}, \ddot{sf}, \bot, \ddot{ic}, \bot)$$

is the corresponding call stack abstraction state.

3.3. Context Concretization

In Section 1.5, the equivalence relations $\equiv$ and $\equiv_{\mathsf{n}}$ help us identify corresponding references in the input states. Likewise, we now make use of similar equivalence relations identifying which parts of the return state correspond to which parts of a call state. The input arguments introduced in Definition 3.3 form the basis of these relations. If an input argument of a state corresponds to a reference of a calling state, then the corresponding data must be identical.

In contrast to the situation in Section 1.5 we may have changed data in one state and outdated data in another state. In such situations we must take care not to consider references equivalent, as the outdated information does not necessarily correspond to the information of the other state. However, as the length of an array cannot be changed, even for arrays that may have changed we know that the length of the array stays the same.

Definition 3.12 ($\equiv$) Let $s = (\langle fr_0, \ldots, fr_n \rangle, h, t, hp, sf, e, ic, \bot)$ with $|s| > 0$ and $\ddot{s} = (\langle \ddot{fr}_0, \ldots, \ddot{fr}_m \rangle, \ddot{h}, \ddot{t}, \ddot{hp}, \ddot{sf}, \bot, \ddot{ic}, \bot)$ be a call state (thus, $|\ddot{s}| > 1$), where $fr_i = (pp_i, lv_i, os_i, ia_i)$ and $\ddot{fr}_i = (\ddot{pp}_i, \ddot{lv}_i, \ddot{os}_i, \ddot{ia}_i)$. Furthermore, let s and $\ddot{s}$ have disjoint sets of references (where only `null` and return addresses may be used in both states). Let $\equiv \subseteq$ References $\times$ References be the smallest equivalence relation which satisfies the following conditions:

(i) $\forall(\lambda, \gamma, \checkmark/\lightning) \in ia_n : \ddot{s} \in \mathrm{dom}(\gamma) \to \lambda \equiv \gamma(\ddot{s})$

(ii) if $r \equiv r'$, $\{h_r(r) = f, h_{r'}(r') = f'\} \subseteq$ Instances, and $r\checkmark$ and $r'\checkmark$ or $s_r = s_{r'}$, then $f(v) \equiv f'(v)$ for all $v \in \mathrm{dom}(f) \cap \mathrm{dom}(f')$

(iii) if $r \equiv r'$, $\{h_r(r) = (i_l, f), h_{r'}(r') = (i'_l, f')\} \subseteq$ Arrays, and $r\checkmark$ and $r'\checkmark$ or $s_r = s_{r'}$, then $f(i) \equiv f'(i)$ for all $i \in \mathrm{dom}(f) \cap \mathrm{dom}(f')$

(iv) if $r \equiv r'$ and $\{h_r(r) = (i_l, f), h_{r'}(r') = (i'_l, f')\} \subseteq$ Arrays, then $i_l \equiv i'_l$

We now illustrate Definition 3.12 using an example.

Example 3.4 Consider the two states from Fig. 3.13. State A is a call state where we created input arguments for the two references r_1 and r_2. State B is a state which resulted out of the call shown in state A, i.e., the only stack frame of B is in the same method as the topmost stack frame of A.

The input arguments in B indicate that r_3 corresponds to r_1 in A, and r_4 corresponds to r_2. Thus, we get $r_1 \equiv r_3$ and $r_2 \equiv r_4$.

A

$(r_1, (A \mapsto r_1), \checkmark)$
$(r_2, (A \mapsto r_2), \checkmark)$
$\langle 0 \mid \mathsf{one} : r_1 \mid \varepsilon \rangle$
$\langle 23 \mid \mathsf{one} : r_2 \mid r_1 \rangle$

$r_1 : \mathsf{List}(\mathsf{next} : r_2)$
$r_2 : \mathsf{List}(\mathsf{next} : r_1)$

B

$(r_3, (A \mapsto r_1), \lightning)$
$(r_4, (A \mapsto r_2), \checkmark)$
$\langle 42 \mid \mathsf{one} : r_3 \mid \varepsilon \rangle$

$r_3 : \mathsf{List}(\mathsf{value} : i_1, \mathsf{next} : r_3)$
$r_4 : \mathsf{List}(\mathsf{value} : i_2, \mathsf{next} : r_3)$
$i_1 : 0$
$i_2 : 1$

Figure 3.13.: Two states illustrating Definition 3.12

Furthermore, the input argument for r_3 indicates a (possible) change. Because of this we do not identify r_2 and r_3 as being equivalent, although we have $r_1 \equiv r_3$ and the referenced objects have r_2 and r_3 as next successors, respectively.

The intuition behind this restriction is that r_3 and r_1 really represent the same object in the heap, but at different points in time. While for example the type information of this object cannot be changed during execution (which we will exploit lateron), the invoked method may have written different values in the fields of r_3. Indeed, while in A the list represented by r_1 has exactly two different elements, in B the corresponding list now only contains a single element (which forms a cycle). Thus, we must not take information available through next into account when intersecting the values r_1 and r_3.

As in Section 1.5, we now extend $\equiv$ to $\equiv_{\mathsf{n}}$. While the definition of $\equiv_{\mathsf{n}}$ does not differ from the definition in Chapter 1, the relations $\twoheadrightarrow$ and $\multimap$ used in it need to be adapted such that the effects of changed references are handled as intended. For this, we first define when an equivalence class is marked as changed. The idea is that we may not consider information which is part of a changed equivalence class, if the information is only available in the call state (and, thus, possibly is outdated).

Definition 3.14 (Changed Equivalence Class) Let $[r]_\equiv$ be an equivalence class. We write $[r]^{\lightning}_{\equiv}$ if there is a reference $r' \in [r]_\equiv$ with $r' \lightning$. If no such r' exist, we write $[r]^{\checkmark}_{\equiv}$.

We now define $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$ to indicate that with the knowledge in $\equiv$ we know that when starting in a reference equivalent to r' and following the edges described by τ (where it is allowed to continue with an equivalent reference in each step), one ends in a reference equivalent to r. As write accesses may delete a path segment only present in $\ddot{s}$, in this case we take care to only consider information in s.

Definition 3.15 ($\overset{\tau}{\underset{\equiv}{\twoheadrightarrow}}$) Let $s, \ddot{s}, \equiv$ as in Definition 3.12. Let r be a reference with $h_r(r) \in \textsc{Instances} \cup \textsc{Arrays}$. Then we have $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$ iff one of the following conditions is met.

- We have $r' \equiv r$ and $\tau = \varepsilon$.
- We have $s_{r'}|_{\pi'} = r' \overset{\tau'}{\underset{\equiv}{\twoheadrightarrow}} r_p = s_{r_p}|_{\pi_p}$, $s_{r_p}|_{\pi_p \tau''} \equiv r$ with $\tau'' \neq \varepsilon$, and $\tau = \tau'\tau''$. If $[r']^{\not{\downarrow}}_{\equiv}$, we also have $s_{r'} = s$. If $[r_p]^{\not{\downarrow}}_{\equiv}$, we also have $s_{r_p} = s$.

If the equivalence relation $\equiv$ is clear from the context, we just write $r' \overset{\tau}{\twoheadrightarrow} r$ instead of $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$.

As we have $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$ iff $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} \mathring{r}$ for all $\mathring{r} \equiv r$, we define that $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} [r]_{\equiv}$ holds if $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$.

Using $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$ we now extend $r' \overset{\tau}{\rightarrowtriangle} r$ as in Definition 1.30 to the setting of this chapter. As we need to deal with invalidated information in $\ddot{s}$, and we have to consider several different equivalence classes, the previous definition of $r' \rightarrowtriangle r$ needs to be adapted accordingly.

We use $r' \overset{\tau}{\rightarrowtriangle} r$ to describe that r' is an abstract predecessor of r in the sense that one cannot continue from r' using τ. Thus, certain properties of r need to be expressed using heap predicates for r'. As an example, if we know that r is cyclic, we need to have $r'\circlearrowleft$.

Definition 3.16 ($\boldsymbol{r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r}$) Let $s_{r'}|_{\pi'} = r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$ where $\tau \neq \varepsilon$ and $\pi' = \overline{\pi'\tau}_{s_{r'}}$. Then we have $r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r$.

As we have $r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r$ iff $r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} \mathring{r}$ for all $\mathring{r} \equiv r$, we define that $r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} [r]_{\equiv}$ holds if $r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r$.

If the equivalence relation $\equiv$ is clear from the context, we just write $r' \overset{\tau}{\rightarrowtriangle} r$ instead of $r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r$. Furthermore, we define that $r' \rightarrowtriangle r$ holds if we have $r' \overset{\tau}{\rightarrowtriangle} r$ for any $\tau \neq \varepsilon$.

From now on we use $r' \rightarrowtriangle r$ as in Definition 3.16 and override the previous Definition 1.30. With those updated definitions, we now define $\equiv_{\mathsf{n}}$.

Definition 3.17 ($\equiv_{\mathsf{n}}$) Let $s, \ddot{s}, \equiv$ as in Definition 3.12. We define $\equiv_{\mathsf{n}}$ based on $\equiv$ as in Definition 1.31, where we use the updated Definitions 3.15 and 3.16.

As in Chapter 1, the information about equivalent references may be conflicting.

Definition 3.18 (Conflicts) Let $s, \ddot{s}, \equiv_{\mathsf{n}}$ as defined in Definition 3.17. If there is a reference $r \equiv_{\mathsf{n}} \mathsf{null}$ where $r \neq \mathsf{null}$ and $r?$ is missing or $h_r(r) \in \textsc{Arrays}$ or $h_r(r) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) \neq \emptyset$, then the context concretization of s and $\ddot{s}$ does not exist.

We also redefine $r' \overset{\tau}{\multimap}\!\!\in r$ using Definitions 3.16 and 3.17.

Definition 3.19 ($r' \overset{\tau}{\multimap}\!\!\in r$) Let $\equiv_{\mathsf{n}}$ as in Definition 3.17. We define that $r' \overset{\tau}{\multimap}\!\!\in r$ holds iff $r' \equiv_{\mathsf{n}} r$ with $\tau = \varepsilon$, or $r' \overset{\tau}{\multimap} r$.

As in Chapter 1, we will use ρ and σ to provide references in the state resulting out of context concretization.

Definition 3.20 (ρ, σ) Let $\equiv_{\mathsf{n}}$ as defined in Definition 3.17. We define ρ, σ according to Definition 1.34.

When intersecting the values of two equivalent references, we need to take care that part of that information may be invalid due to side effects. For example, we must not retain the information of the field contents of some object instance in $\ddot{s}$, if in s we possibly have overwritten all field contents of an equivalent reference.

Thus, we extend the definition of $\Cap$ by not only providing the values to intersect, but also expressing if that data is known to be valid. The idea is to only regard field or array index information for the more dominant value. If a value is known to be unchanged ($\checkmark$), its information is only part of the result if the other value is in the same state or it is also known to be unchanged. Thus, as soon as data from s which is marked as changed is intersected with any information in $\ddot{s}$, only the information of s is retained.

Definition 3.21 ($\Cap^{\checkmark}_{\lightning}$) Let $\textsc{Values}$ as in Definition 1.36. Then we have

$$\Cap^{\checkmark}_{\lightning} : (\textsc{Values} \times \{\checkmark, \lightning\})^2 \to \textsc{Values} \times \{\checkmark, \lightning\}$$

and define $(v, v^{\checkmark}_{\lightning}) \Cap^{\checkmark}_{\lightning} (v', v'^{\checkmark}_{\lightning})$ as follows.

$(v, v^{\checkmark}_{\lightning}) \Cap^{\checkmark}_{\lightning} (v', v'^{\checkmark}_{\lightning})$	condition
$(\sigma(i_l), f\sigma), \lightning)$	$\{v = (i_l, f), v'\} \subseteq \textsc{Arrays} \wedge v^{\checkmark}_{\lightning} = \lightning \wedge v'^{\checkmark}_{\lightning} = \checkmark$
$(\sigma(i_l), f'\sigma), \lightning)$	$\{v, v' = (i'_l, f')\} \subseteq \textsc{Arrays} \wedge v^{\checkmark}_{\lightning} = \checkmark \wedge v'^{\checkmark}_{\lightning} = \lightning$
$(\sigma(i_l), (f \cup f')\sigma), v^{\checkmark}_{\lightning})$	$\{v = (i_l, f), v' = (i'_l, f')\} \subseteq \textsc{Arrays} \wedge v^{\checkmark}_{\lightning} = v'^{\checkmark}_{\lightning}$
$(\sigma(i_l), f\sigma), v^{\checkmark}_{\lightning})$	$v = (i_l, f) \in \textsc{Arrays} \wedge v' = f' \in \textsc{Instances}$ $\wedge \operatorname{dom}(f') = \emptyset \wedge (v^{\checkmark}_{\lightning} = \lightning \vee v^{\checkmark}_{\lightning} = v'^{\checkmark}_{\lightning})$
$(\sigma(i_l), f'\sigma), \lightning)$	$v = (i_l, f) \in \textsc{Arrays} \wedge v' = f' \in \textsc{Instances}$ $\wedge \operatorname{dom}(f') = \emptyset \wedge v^{\checkmark}_{\lightning} = \checkmark \wedge v'^{\checkmark}_{\lightning} = \lightning$
$(※, \lightning)$	$v \in \textsc{Arrays} \wedge v' = f' \in \textsc{Instances} \wedge \operatorname{dom}(f') \neq \emptyset$
$(\sigma(i'_l), f'\sigma), v'^{\checkmark}_{\lightning})$	$v = f \in \textsc{Instances} \wedge v' = (i'_l, f') \in \textsc{Arrays}$ $\wedge \operatorname{dom}(f) = \emptyset \wedge (v'^{\checkmark}_{\lightning} = \lightning \vee v^{\checkmark}_{\lightning} = v'^{\checkmark}_{\lightning})$
$(\sigma(i'_l), f\sigma), \lightning)$	$v = f \in \textsc{Instances} \wedge v' = (i'_l, f') \in \textsc{Arrays}$ $\wedge \operatorname{dom}(f) = \emptyset \wedge v^{\checkmark}_{\lightning} = \lightning \wedge v^{\checkmark}_{\lightning} = \checkmark$
$(※, \lightning)$	$v = f \in \textsc{Instances} \wedge v' \in \textsc{Arrays} \wedge \operatorname{dom}(f) \neq \emptyset$
$(f\sigma, \lightning)$	$\{v = f, v'\} \subseteq \textsc{Instances} \wedge v^{\checkmark}_{\lightning} = \lightning \wedge v'^{\checkmark}_{\lightning} = \checkmark$
$(f'\sigma, \lightning)$	$\{v, v' = f'\} \subseteq \textsc{Instances} \wedge v^{\checkmark}_{\lightning} = \checkmark \wedge v'^{\checkmark}_{\lightning} = \lightning$
$((f \cup f')\sigma, v^{\checkmark}_{\lightning})$	$\{v = f, v' = f'\} \subseteq \textsc{Instances} \wedge v^{\checkmark}_{\lightning} = v'^{\checkmark}_{\lightning}$
$(v \Cap v', \lightning)$	otherwise (with $\Cap$ as in Definition 1.36)

In order to successfully perform context concretization, we only need to regard a state s that corresponds to the call state $\ddot{s}$. In Definition 3.22 we check the corresponding conditions, so that in the definition of context concretization we know that s and $\ddot{s}$ are valid input states. In Definition 3.22(i–iii) we compare some fundamental properties of the states, namely that the invoked method corresponds to the method in s and that the initialization state allows for a possible evaluation from $\ddot{s}$ to s (i.e., no class which was initialized in $\ddot{s}$ is not initialized anymore in s). In Definition 3.22(iv), which corresponds to Definition 1.10(d), we take care that only states which may represent the same computation are considered in context concretization. Finally, in Definition 3.22(v,vi) we ensure that the input arguments created for $\ddot{s}$, which are needed for any context concretization with $\ddot{s}$, also are represented in s.

Definition 3.22 (s and $\ddot{s}$ are valid for context concretization) Let s and $\ddot{s}$ be states as defined in Definition 3.12. We define that s and $\ddot{s}$ are valid for context concretization only if the following conditions are met:

(i) The opcode pp_n is in the same method as the opcode $\ddot{pp}_0$.

(ii) For all classes cl with $\ddot{ic}(cl) = \text{YES}$ we have $ic(cl) = \text{YES}$.

(iii) For all classes cl with $\ddot{ic}(cl) = \text{RUNNING}$ we have $ic(cl) \in \{\text{RUNNING}, \text{YES}\}$.

(iv) For each class $[r]_{\equiv_n}$ where r is a return address we have $r' = r$ for all $r' \in [r]_{\equiv_n}$.

(v) For each input argument in $\ddot{ia}_0$ we have a corresponding input argument in s: $\forall(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0 \rightarrow \exists(\lambda, \gamma, \checkmark/\lightning) \in ia_n : \ddot{\gamma} = \gamma$.

(vi) For every input argument for $\ddot{s}$ we have a corresponding input argument in $\ddot{ia}_0$: $\forall(\lambda, \gamma, \checkmark/\lightning) \in ia_n : \ddot{s} \in \text{dom}(\gamma) \rightarrow \exists(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0 : \gamma = \ddot{\gamma}$

With these auxiliary definitions we can now define the context concretization of a state s with a call state $\ddot{s}$, giving us a state $\tilde{s}$ that defines how we return from a recursive method. Here, we closely follow the idea of state intersection to combine the information available in s and $\ddot{s}$. For the upcoming correctness proofs we do not only show context concretization if s is a return state, but also for arbitrary states. However, in the actual analysis we only use context concretization for s if s is a return state.

In addition to the concepts already explained for state intersection, the main idea of context concretization is to combine the two states s and $\ddot{s}$. Here, the call stack of $\tilde{s}$ is built using the stack frames of s on top and all but the topmost stack frame of $\ddot{s}$ below that. This corresponds to the idea that $\ddot{s}$ represents the context of the method invocation, and s contains information about the invoked method.

In contrast to Chapter 1 we may also have input arguments in the states. Indeed, if $\tilde{s}$ is the state resulting out of context concretization of s and $\ddot{s}$, there may be a call state $\ddot{s}'$ so that we may also need to compute the context concretization of a state resulting out of $\tilde{s}$ (then in the role of the return state) and $\ddot{s}'$. Thus, we need to add input arguments to $\tilde{s}$ which can be used in such situations.

The input arguments in s are only used to compute $\tilde{s}$, thus these are dropped and not represented anymore in $\tilde{s}$. Likewise we do not need to consider the input arguments in the topmost stack frame of $\ddot{s}$.

Let $m_{\ddot{s}}$ be the method represented in the lowest stack frame of $\ddot{s}$. Likewise, let m_s be the method represented in the lowest stack frame of s, i.e., m_s is the method invoked from $\ddot{s}$. The lowest stack frame of $\ddot{s}$ may contain input arguments which may be used in another context concretization, when $m_{\ddot{s}}$ returns. Thus, in $\tilde{s}$ we must take care to add corresponding input arguments. Let m be any such method where $m_{\ddot{s}}$ is invoked. Evaluation of m_s may have caused side effects which are observable from $m_{\ddot{s}}$. Furthermore, there may be side effects in m_s that also are observable from m. Thus, we need to update the input arguments added to the lowest stack frame of $\tilde{s}$ such that these changes are represented. Because of that, if we find an input argument in the topmost stack frame of $\ddot{s}$ for which the corresponding input argument in s is marked as changed, we look for input arguments in the lowest stack frame of $\ddot{s}$ which may reach the input argument in the topmost stack frame. If such an input argument exists, we use its information to create an input argument in the lowest stack frame of $\tilde{s}$ which is marked as changed. Thus, side effects initially observed in s are propagated to $\tilde{s}$ and, consequently, also to states resulting out of further context concretizations.

The computation of the heap predicates in $\tilde{s}$ is similar to the ideas used for state intersection. However, as there may be outdated information in $\ddot{s}$ not all heap predicates may be considered when creating $\tilde{s}$. For example in $\ddot{s}$ we may have a reference r for which no $r\circlearrowright_F$ exists. Thus, in the setting of state intersection we can be sure that r indeed is acyclic (or all cycles are explicitly represented). However, if a reference r' in s which is equivalent to r has been changed, we do not know for sure that the reference representing r and r' in $\tilde{s}$ is acyclic. Instead, in the case of changes we must only regard the information available in s.

Definition 3.23 (Context Concretization) Let $s, \ddot{s}, \equiv_{\mathsf{n}}, \rho, \sigma$ as defined in Definitions 3.17 and 3.20. Let $fr_i = (pp_i, lv_i, os_i, ia_i)$ for $0 \le i \le n$ be the stack frames of s and let $\ddot{fr}_i = (\ddot{pp}_i, \ddot{ia}_i, \ddot{os}_i, \ddot{ia}_i)$ for $0 \le i \le m$ be the stack frames of $\ddot{s}$. Let s and $\ddot{s}$ be valid in the sense of Definition 3.22, and let there be no conflict as in Definition 3.18.

We define a function cc: $\textsc{States} \times \textsc{States} \to \textsc{States} \cup \{※\}$. If the context concretization as described below does not exist, then $\mathrm{cc}(s, \ddot{s}) = ※$.

We define $\mathrm{cc}(s, \ddot{s}) = \tilde{s}$ with

$$\tilde{s} = (\langle \widetilde{fr}_0, \ldots, \widetilde{fr}_n, \widetilde{fr}_{n+1}, \ldots, \widetilde{fr}_{n+m} \rangle, \tilde{h}, \tilde{t}, \widetilde{hp}, sf\sigma, \tilde{e}, ic, \bot)$$

Call Stack We define $\widetilde{fr}_i = (pp_i, lv_i\sigma, os_i\sigma, \emptyset)$ for $0 \le i \le n$, $\widetilde{fr}_i = (\ddot{pp}_i, \ddot{lv}_i\sigma, \ddot{os}_i\sigma, \emptyset)$ for $n+1 \le i < n+m$, and $\widetilde{fr}_{n+m} = (\ddot{pp}_m, \ddot{lv}_m\sigma, \ddot{os}_m\sigma, \widetilde{ia})$ with $\widetilde{ia}$ as follows.

If we have $(\ddot{\lambda}, \ddot{\gamma}, \lightning) \in \ddot{ia}_m$, then we also have $(\ddot{\lambda}\sigma, \ddot{\gamma}, \lightning) \in \widetilde{ia}$. Otherwise, assume we have $(\ddot{\lambda}, \ddot{\gamma}, \checkmark) \in \ddot{ia}_m$. If there is $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $[\ddot{\lambda}']^{\lightning}_{\equiv_{\mathsf{n}}}$ and $\ddot{\lambda} \rightsquigarrow \ddot{\lambda}'$, then we have $(\ddot{\lambda}\sigma, \ddot{\gamma}, \lightning) \in \widetilde{ia}$. Otherwise, we have $(\ddot{\lambda}\sigma, \ddot{\gamma}, \checkmark) \in \widetilde{ia}$.

Exception Using σ we define $\tilde{e} = \bot$ if $e = \bot$, otherwise $\tilde{e} = \sigma(e)$.

Types We now define the type component $\tilde{t}$. Let $r \in \textsc{References}(s) \cup \textsc{References}(\ddot{s})$ where $r = \mathsf{null}$ or the heap maps r to a value in $\textsc{Instances} \cup \textsc{Arrays}$:

$$\tilde{t}(\sigma(r)) = \bigcap_{r' \in [r]_{\equiv_{\mathsf{n}}}} t_{r'}(r')$$

Heap For $r \in \textsc{References}(s) \cup \textsc{References}(\ddot{s})$ with $r \notin [\mathsf{null}]_{\equiv_{\mathsf{n}}}$ and r is no return address we now define the heap component (cf. Definition 3.21). We define $\tilde{h}(\sigma(r)) = r_\cap$ with

$$(r_\cap, \checkmark/\lightning) = \bigcap_{r' \in [r]_{\equiv_n}}^{\checkmark/\lightning} (h_{r'}(r'), r'^{\checkmark}_{\lightning})$$

Here, we have $r'^{\checkmark}_{\lightning} = \lightning$ if $[r']^{\lightning}_{\equiv_n}$ and $s_{r'} = s$, and $r'^{\checkmark}_{\lightning} = \checkmark$ otherwise.

If for any reference r the intersection results in $\tilde{h}(r) = ※$, then $\tilde{s}$ does not exist.

Heap Predicates Finally, we define the heap predicates $\widetilde{hp}$. Let $r \neq r'$ be two references with $\{\tilde{h}(\sigma(r)), \tilde{h}(\sigma(r'))\} \subseteq \textsc{Instances} \cup \textsc{Arrays}$:

(a) We add $\sigma(r)?$ if for all $r' \in [r]_{\equiv_n}$ we have $r'?$.

(b) We add $\sigma(r) =^? \sigma(r')$ if we have $[r]_{\equiv_n} \neq [r']_{\equiv_n}$, and $r_i \in [r]_{\equiv_n}, r'_i \in [r']_{\equiv_n}$ with $r_i =^? r'_i$, and for all $r_i \in [r]_{\equiv_n}, r'_i \in [r']_{\equiv_n}$ we either have $r_i =^? r'_i$ or r_i, r'_i are not in the same state. If we have $r_a \rightarrow [\pi]_{\equiv_n}$ and $r_b \rightarrow [\pi']_{\equiv_n}$ with $s_{r_a} = s_{r_b}$ we also demand that $r_a \searrow\!\!\swarrow r_b$ exists. Furthermore, if $s_{r'}|_{\pi_a} = r_a \xrightarrow{\tau_a} [\pi]_{\equiv_n}$, we need to have $s_{r'}|_{\overline{\pi_a \tau_a}} \searrow\!\!\swarrow r'$. Similarly, if $s_r|_{\pi_b} = r_b \xrightarrow{\tau_b} [\pi']_{\equiv_n}$, we need to have $s_r|_{\overline{\pi_b \tau_b}} \searrow\!\!\swarrow r$.

(c) We add $\sigma(r) \searrow\!\!\swarrow \sigma(r')$, if $[r]^{\lightning}_{\equiv_n}$ or $[r']^{\lightning}_{\equiv_n}$, there exist $r_1 \in [r]_{\equiv_n}, r_2 \in [r']_{\equiv_n}$ with $r_1 \searrow\!\!\swarrow r_2$ and $s_{r_1} = s_{r_2} = s$, and for all $r_1 \in [r]_{\equiv_n}, r_2 \in [r']_{\equiv_n}$ with $s_{r_1} = s_{r_2} = s$ we have $r_1 \searrow\!\!\swarrow r_2$.

(d) We add $\sigma(r) \searrow\!\!\swarrow \sigma(r')$, if $[r]^{\checkmark}_{\equiv_n}$ and $[r']^{\checkmark}_{\equiv_n}$, there exist $r_1 \in [r]_{\equiv_n}, r_2 \in [r']_{\equiv_n}$ with $r_1 \searrow\!\!\swarrow r_2$, and for all $r_1 \in [r]_{\equiv_n}, r_2 \in [r']_{\equiv_n}$ we have $r_1 \searrow\!\!\swarrow r_2$ or r_1, r_2 are not in the same state.

(e) We add $\sigma(r) \searrow\!\!\swarrow \sigma(r)$, if $[r]^{\lightning}_{\equiv_n}$ and there exists $r' \in [r]_{\equiv_n}$ with $s_{r'} = s$ and $r' \searrow\!\!\swarrow r'$, and for all $r' \in [r]_{\equiv_n}$ with $s_{r'} = s$ we have $r' \searrow\!\!\swarrow r'$.

(f) We add $\sigma(r) \searrow\!\!\swarrow \sigma(r)$, if $[r]^{\checkmark}_{\equiv_n}$ and there exists $r' \in [r]_{\equiv_n}$ with $r' \searrow\!\!\swarrow r'$, and for all $r' \in [r]_{\equiv_n}$ we have $r' \searrow\!\!\swarrow r'$.

(g) We add $\sigma(r) \circlearrowleft_F$ with $F = \bigcup_i F_i$, if $[r]^{\lightning}_{\equiv_n}$ and there exists $r' \in [r]_{\equiv_n}$ with $s_{r'} = s$ and $r' \circlearrowleft_{F'}$, and for all $r' \in [r]_{\equiv_n}$ with $s_{r'} = s$ we have $r' \circlearrowleft_{F_i}$.

(h) We add $\sigma(r) \circlearrowleft_F$ with $F = \bigcup_i F_i$, if $[r]^{\checkmark}_{\equiv_n}$ and there exists $r' \in [r]_{\equiv_n}$ with $r' \circlearrowleft_{F'}$, and for all $r' \in [r]_{\equiv_n}$ we have $r' \circlearrowleft_{F_i}$.

We now show the basic concepts of Definition 3.23 using an example.

Example 3.5 Consider the states shown in Fig. 3.24 where we apply context concretization of B with A (which is a call state). We first check that the two states are valid for context concretization as in Definition 3.22. This is the case, as we assume the opcodes of the topmost stackframes are in the same method. Furthermore, for every

A	B	C
$(r_1, (A \mapsto r_1), \checkmark)$	$(r_3, (A \mapsto r_1), \lightning)$	$\langle 42 \mid \mathsf{one} : r_6 \mid \varepsilon \rangle$
$(r_2, (A \mapsto r_2), \checkmark)$	$(r_4, (A \mapsto r_2), \checkmark)$	$(r_6, (D \mapsto r_5), \lightning)$
$\langle 0 \mid \mathsf{one} : r_1 \mid \varepsilon \rangle$	$\langle 42 \mid \mathsf{one} : r_3 \mid \varepsilon \rangle$	$\langle 23 \mid \mathsf{one} : r_7 \mid r_6 \rangle$
$(r_1, (D \mapsto r_5), \checkmark)$		
$\langle 23 \mid \mathsf{one} : r_2 \mid r_1 \rangle$		
$r_1 \colon \mathsf{List}(\mathsf{value} \colon i_1, \mathsf{next} \colon r_2)$	$r_3 \colon \mathsf{List}(\mathsf{value} \colon i_2, \mathsf{next} \colon r_3)$	$r_6 \colon \mathsf{List}(\mathsf{value} \colon i_4, \mathsf{next} \colon r_6)$
$r_2 : \mathsf{List}(\mathsf{next} : r_1)$	$r_4 \colon \mathsf{List}(\mathsf{value} \colon i_3, \mathsf{next} \colon r_3)$	$r_7 \colon \mathsf{List}(\mathsf{value} \colon i_5, \mathsf{next} \colon r_6)$
$i_1 : 0$	$i_2 : [0,1]$	$i_4 : [0,1]$
	$i_3 : 1$	$i_5 : 1$

Figure 3.24.: Two states illustrating Definition 3.12

input argument in A we also have a corresponding input argument in B, and for each input argument of B referencing data in A we also have an input argument in A.

We also have no conflict (Definition 3.18), thus we now construct the resulting state $\mathrm{cc}(A, B) = C$, shown in Fig. 3.24. The call stack is constructed by taking the frames of B, but removing the input arguments. Below we add the lower stack frame of A, where we modify the input arguments.

We have an input argument for r_1 in A, which is created for a call state named D (not shown in this example). As we have $r_1 \equiv_{\mathsf{n}} r_3$ and we have $r_3 \lightning$, the added input argument is also marked as possibly changed.

The values in C are created based on Definition 3.21. We start with $r_2 \equiv_{\mathsf{n}} r_4$. As both these references are left unchanged, we use the field information of both referenced instances and obtain $\mathsf{List}(\mathsf{next} \colon r_7, \mathsf{value} \colon i_5)$. Here, i_5 is the reference used for $[i_3]_{\equiv_{\mathsf{n}}}$.

We also intersect $r_1 \equiv_{\mathsf{n}} r_3$. However, we have $r_3 \lightning$. Thus, the field information of r_1 is not used in this intersection. Instead, we just consider the values of r_3. This results in $\mathsf{List}(\mathsf{value} \colon i_4, \mathsf{next} \colon 6)$.

As there are no heap predicates in A nor B, construction of C is now finished.

This definition now allows us to reason about instances of states even if only parts of the call stack are represented. We may have $s' \sqsubseteq s$ where the call stack of s is lower than the call stack of s'. Here, the main idea is to repeatedly apply context concretization to s until s' and the resulting state have a call stack of the same height. If we obtain $\tilde{s}$ based on s with $|\tilde{s}| = |s'|$, we can check $s' \sqsubseteq \tilde{s}$ similar to how we did it in Chapter 1. The only relevant change is that, for $s' \sqsubseteq s$ with $|s'| = |s|$, the two states must have equivalent input arguments. Furthermore, if any input argument in s' is marked as changed, the corresponding input argument in s must also be marked as changed. This idea results in the following recursive definition.

Definition 3.25 ($\sqsubseteq$) Let s and s' be two states. If $|s'| > |s|$, we have $s' \sqsubseteq s$ iff there is a call state $\ddot{s}$ such that $\mathrm{cc}(s, \ddot{s}) = \tilde{s}$ and $s' \sqsubseteq \tilde{s}$.

Otherwise, if $|s'| \leq |s|$, we have $s' \sqsubseteq s$ iff the conditions of Definition 1.10(a–r) are satisfied (which also means we have $|s'| = |s|$) and the following additional condition holds:

(s) For each $(\lambda', \gamma', \checkmark/\lightning) \in ia'_i$ we have $(\lambda, \gamma, \checkmark/\lightning) \in ia_i$ with $\gamma' = \gamma$, and for each $(\lambda', \gamma', \checkmark/\lightning) \in ia_i$ we have $(\lambda', \gamma', \checkmark/\lightning) \in ia'_i$ with $\gamma = \gamma'$.

For each $(\lambda', \gamma', \lightning) \in ia'_i$ we have $(\lambda, \gamma, \lightning) \in ia_i$ with $\gamma' = \gamma$.

Intuitively for two states s' and s with $|s'| = |s|$ and $s' \sqsubseteq s$ we expect that the more abstract state s does not contain any state position which does not exist in s'.

Lemma 3.6 (Positions in Instances) For states $s' \sqsubseteq s$ with $|s'| = |s|$ we have $\mathrm{SPos}(s') \supseteq \mathrm{SPos}(s)$.

Proof. According to Definition 3.25(s) for input arguments we have the same set of positions of the form $\mathrm{IA}_{i,\gamma}$. The remainder of the proof corresponds to the proof of Lemma 1.3. □

3.4. Stability of $\sqsubseteq$ Under Context Concretization

As the second part of Definition 3.25 does not differ much from Definition 1.10, it is not hard to see its use. However, in the first part we take context concretization into account when defining state instances. Thus, we need to understand that applying context concretization to states retains the relation defined by $\sqsubseteq$.

Theorem 3.7 (Stability of $\sqsubseteq$) Let $s, s' \in \mathrm{States}$ with $|s'| = |s|$, $s' \sqsubseteq s$, and let $\ddot{s}$ be a call state. If we have $\mathrm{cc}(s', \ddot{s}) = \tilde{s}'$, then we also have $\mathrm{cc}(s, \ddot{s}) = \tilde{s}$ with $\tilde{s}' \sqsubseteq \tilde{s}$.

The proof of Theorem 3.7 actually is quite involved and is split into several auxiliary lemmas. The main reason for this is that five different states $(s', s, \ddot{s}, \tilde{s}', \tilde{s})$ are involved, in addition to that we need to consider two equivalence relations, and also deal with complications due to side effects (i.e., the states may represent outdated information). Finally, the proofs already done for state intersection have to be adapted.

In most of the upcoming lemmas, we consider a very specific situation. Thus, we first define this situation with all relevant symbols so that we can simply refer to this definition later.

Definition 3.26 (Situation) Let $s', s, \ddot{s}, \tilde{s}', \tilde{s}$ be states where $\ddot{s}$ is a call state, $s' \sqsubseteq s$ with $|s'| = |s|$, $\mathrm{cc}(s, \ddot{s}) = \tilde{s}$, and $\mathrm{cc}(s', \ddot{s}) = \tilde{s}'$. We assume that, apart from null and return addresses, $s, s', \ddot{s}$ have disjoint references. Furthermore we have $\equiv'$ constructed for s' and $\ddot{s}$, and $\equiv$ constructed for s and $\ddot{s}$ (as defined in Definition 3.12). Let $\equiv'_{\mathsf{n}}$ and $\equiv_{\mathsf{n}}$ be the corresponding extensions according to Definition 3.17. Based on Definition 3.20, let σ', ρ' be used for $\tilde{s}'$, likewise let σ, ρ be used for $\tilde{s}$.

For $s_r \in \{s, \ddot{s}\}$ we define

$$\overleftarrow{s_r} := \begin{cases} \ddot{s} & \text{if } s_r = \ddot{s} \\ s' & \text{if } s_r = s \end{cases}$$

Note that we have $|s_r| = |\overleftarrow{s_r}|$ and $\overleftarrow{s_r} \sqsubseteq s_r$. We also extend this notation to $\overleftarrow{t_r}$ and $\overleftarrow{h_r}$.

First, we show how the equivalence relation $\equiv_{\mathsf{n}}$ created for a state s and a call state $\ddot{s}$ relates to $\equiv'_{\mathsf{n}}$ created for s' and the same call state $\ddot{s}$. To simplify the proof, we first show the claim for $\equiv$ instead of $\equiv_{\mathsf{n}}$.

Lemma 3.8 Let $s', s, \equiv', \equiv$ as in Definition 3.26.

Then for all $s_r|_\pi = r \equiv r' = s_{r'}|_{\pi'}$ we have $\overleftarrow{s_r}|_\pi \equiv' \overleftarrow{s_{r'}}|_{\pi'}$.

Proof. First we show that it suffices to have a single pair of positions $\pi \neq \pi'$ with $s_r|_\pi = r \equiv r' = s_{r'}|_{\pi'}$ and $\overleftarrow{s_r}|_\pi \equiv' \overleftarrow{s_{r'}}|_{\pi'}$ to show the claim. Then, for any positions $\mathring{\pi}, \mathring{\pi}'$ with $s_r|_{\mathring{\pi}} = r$ and $s_{r'}|_{\mathring{\pi}'} = r'$, with $s' \sqsubseteq s$ and Definition 3.25(k) we also have $\overleftarrow{s_r}|_\pi = \overleftarrow{s_r}|_{\mathring{\pi}}$ and $\overleftarrow{s_{r'}}|_{\pi'} = \overleftarrow{s_{r'}}|_{\mathring{\pi}'}$. Thus, with $\overleftarrow{s_r}|_{\mathring{\pi}} = \overleftarrow{s_r}|_\pi \equiv' \overleftarrow{s_{r'}}|_{\pi'} = \overleftarrow{s_{r'}}|_{\mathring{\pi}'}$ we also have $\overleftarrow{s_r}|_{\mathring{\pi}} \equiv' \overleftarrow{s_{r'}}|_{\mathring{\pi}'}$.

We show the claim by using an induction. Assume we have $r \equiv r'$ because $r = r'$. If $s_r = s_{r'} = \ddot{s}$, the proof is trivial. If $s_r = s_{r'} = s$, for any positions π, π' with $s|_\pi = s|_{\pi'} = r = r'$ with Definition 3.25(k) and $s' \sqsubseteq s$ we also have $s'|_\pi = s'|_{\pi'}$, thus $s'|_\pi \equiv' s'|_{\pi'}$. If $s_r = s, s_{r'} = \ddot{s}$, consider any positions π, π' with $s|_\pi = r$ and $\ddot{s}|_{\pi'} = r'$. With Definition 3.25(d,h) and $s' \sqsubseteq s$ we also have $s|_\pi = s'|_\pi = r$, thus $s'|_\pi \equiv' \ddot{s}|_{\pi'}$. The case with $s_r = \ddot{s}, s_{r'} = s$ is analogous.

Next, we consider $r \equiv r'$ because we have $r = \lambda$ and $(\lambda, \gamma, \checkmark/\lightning) \in ia_n$ with $\gamma(\ddot{s}) = r'$. Thus, we have $s|_{\mathrm{IA}_{n,\gamma}} = r = s|_\pi$. According to Definition 3.25(s) we also have $(\lambda', \gamma', \checkmark/\lightning) \in ia'_n$ with $\gamma' = \gamma$, thus also $\gamma'(\ddot{s}) = r' = \gamma(\ddot{s})$ and $s'|_{\mathrm{IA}_{n,\gamma'}} \equiv' r'$. With $\overleftarrow{s_r} = s'$, $\overleftarrow{s_{r'}} = \ddot{s}$, $s|_{\mathrm{IA}_{n,\gamma}} = s|_{\mathrm{IA}_{n,\gamma'}} = r$, and $s'|_{\mathrm{IA}_{n,\gamma'}} \equiv' \ddot{s}|_{\pi'} = r'$ the claim follows.

Now consider that we have $r \equiv r'$ because there are r_p, r'_p with $r_p \equiv r'_p$, $h_{r_p}(r_p) = f, h_{r'_p}(r'_p) = f'$, and $f(v) = r, f'(v) = r'$ for some $v \in$ FIELDIDs where $r_p\checkmark$ and $r'_p\checkmark$, or $s_{r_p} = s_{r'_p}$. By induction, we know $\overleftarrow{s}_{r_p}|_\pi \equiv' \overleftarrow{s}_{r'_p}|_{\pi'}$ for all π, π' with $s_{r_p}|_\pi = r_p$ and $s_{r'_p}|_{\pi'} = r'_p$. With Definition 3.25(s) we also know that $\overleftarrow{s}_{r_p}|_\pi\checkmark$ and $\overleftarrow{s}_{r'_p}|_{\pi'}\checkmark$, or $\overleftarrow{s}_{r_p} = \overleftarrow{s}_{r'_p}$. Thus, with Definition 3.25(i) we also have $\overleftarrow{s}_{r_p}|_{\pi v} \equiv' \overleftarrow{s}_{r'_p}|_{\pi' v}$. The cases involving Definition 3.12(iii,iv) are analogous.

Finally, we consider $r \equiv r'$ because we have $r \equiv r_m$ and $r_m \equiv r'$. With $r \equiv r_m$, by induction, we know $\overleftarrow{s}_r|_\pi \equiv' \overleftarrow{s}_r|_{\pi_m}$ for all π, π_m with $s_r|_\pi = r$ and $s_{r_m}|_{\pi_m} = r_m$. Similarly, we also have $\overleftarrow{s}_{r'}|_{\pi'} \equiv' \overleftarrow{s}_{r'}|_{\pi_m}$, where $s_{r'}|_{\pi'} = r'$. Combining this, we get $\overleftarrow{s}_r|_\pi \equiv' \overleftarrow{s}_{r'}|_{\pi'}$. $\square$

Before we extend the proof to $\equiv_{\mathsf{n}}$, we need to reason about the relations $\twoheadrightarrow$ and $\rightarrowtriangle$.

Lemma 3.9 Let $\equiv', \equiv$ as in Definition 3.26.
Then for $s_{r'}|_{\pi'} = r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r = s_r|_\pi$ we have $\overleftarrow{s}_{r'}|_{\pi'} \overset{\tau}{\underset{\equiv'}{\twoheadrightarrow}} \overleftarrow{s}_r|_\pi$.

Proof. We show the claim using an induction. In the base case we have $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$ with $r' \equiv r$. Thus, the claim follows with Lemma 3.8.

Otherwise, we have $r' \overset{\tau'\tau''}{\underset{\equiv}{\twoheadrightarrow}} r$ with $r' \overset{\tau'}{\underset{\equiv}{\twoheadrightarrow}} s_{r_p}|_{\pi_p}, s_{r_p}|_{\pi_p\tau''} \equiv r$. If $[r']^{\lightning}_{\equiv}$, we know $s_{r'} = s$ and $\overleftarrow{s}_{r'} = s'$. If $[r_p]^{\lightning}_{\equiv}$, we know $s_{r_p} = s$ and $\overleftarrow{s}_{r_p} = s'$.

With Lemma 3.6 we know that for $\pi_p\tau' \in \text{SPOS}(s_{r_p})$, $\pi \in \text{SPOS}(s_r)$, and $\pi' \in \text{SPOS}(s_{r'})$ we also have $\pi_p\tau' \in \text{SPOS}(\overleftarrow{s}_{r_p})$, $\pi \in \text{SPOS}(\overleftarrow{s}_r)$, and $\pi' \in \text{SPOS}(\overleftarrow{s}_{r'})$. With Lemma 3.8 we know that $\overleftarrow{s}_{r_p}|_{\pi_p\tau''} \equiv' \overleftarrow{s}_r|_\pi$.

By induction we know $\overleftarrow{s}_{r'}|_{\pi'} \overset{\tau'}{\underset{\equiv'}{\twoheadrightarrow}} \overleftarrow{s}_{r_p}|_{\pi_p}$. We also have $\overleftarrow{s}_{r_p}|_{\pi_p\tau''} \equiv' \overleftarrow{s}_r|_\pi$. Thus, $\overleftarrow{s}_{r'}|_{\pi'\tau'} \underset{\equiv'}{\twoheadrightarrow} \overleftarrow{s}_r|_\pi$. $\square$

Lemma 3.10 Let $\equiv', \equiv$ as in Definition 3.26.
Then for all $s_{r'}|_{\pi'} = r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r = s_r|_\pi$ we have $\overleftarrow{s}_{r'}|_{\overline{\pi'\tau}} \overset{\tau'}{-\!\!\in} \overleftarrow{s}_r|_\pi$ for $\pi'\tau = \overline{\pi'\tau}_{\overleftarrow{s}_{r'}}\tau'$.

Proof. Let $s_{r'}|_{\pi'} = r' \overset{\tau}{\underset{\equiv}{\rightarrowtriangle}} r = s_r|_\pi$ with $\tau \neq \varepsilon$. Thus, we also have $r' \overset{\tau}{\underset{\equiv}{\twoheadrightarrow}} r$. With Lemma 3.9 we have $\overleftarrow{s}_{r'}|_{\pi'} \overset{\tau}{\underset{\equiv'}{\twoheadrightarrow}} \overleftarrow{s}_r|_\pi$. If $[r']^{\lightning}_{\equiv}$, we know $s_{r'} = s$. Thus, if $[r']^{\lightning}_{\equiv'}$ we also have $\overleftarrow{s}_{r'} = s'$.

If $\pi' = \overline{\pi'\tau}_{\overleftarrow{s}_{r'}}$, we have $\overleftarrow{s}_{r'}|_{\pi'} \underset{\equiv'}{\overset{\tau}{\rightarrowtriangle}} \overleftarrow{s}_r|_\pi$. Otherwise we have $\pi'\tau' \in \mathrm{SPos}(\overleftarrow{s}_{r'})$ for some $\varepsilon \neq \tau' \trianglelefteq \tau$ and $\overline{\pi'\tau}_{s_{r'}} = \pi'\tau'$. If $\tau' = \tau$, with Definitions 3.12 and 3.15 we have $\overleftarrow{s}_{r'}|_{\pi'\tau} \equiv' \overleftarrow{s}_r|_\pi$, thus also $\overleftarrow{s}_r|_\pi \equiv_{\mathsf{n}}' \overleftarrow{s}_{r'}|_{\pi'\tau}$. If $\tau' \vartriangleleft \tau$, with Definitions 3.12 and 3.15 we have $\overleftarrow{s}_{r'}|_{\pi'\tau'} \underset{\equiv'}{\overset{\tau''}{\twoheadrightarrow}} \overleftarrow{s}_r|_\pi$ where $\tau = \tau'\tau''$ and also have $\overleftarrow{s}_{r'}|_{\pi'\tau'} \underset{\equiv'}{\overset{\tau''}{\rightarrowtriangle}} \overleftarrow{s}_r|_\pi$. □

Now, as we have shown auxilliary lemmas for $\twoheadrightarrow$ and $\rightarrowtriangle$, we can show that $\equiv_{\mathsf{n}}'$ holds if $\equiv_{\mathsf{n}}$ holds.

Lemma 3.11 Let $\equiv_{\mathsf{n}}', \equiv_{\mathsf{n}}$ as in Definition 3.26.

Then for all $r \equiv_{\mathsf{n}} r'$ and all π, π' with $s_r|_\pi = r$ and $s_{r'}|_{\pi'} = r'$ we have $\overleftarrow{s}_r|_\pi \equiv_{\mathsf{n}}' \overleftarrow{s}_{r'}|_{\pi'}$.

Proof. As we already have shown Lemma 3.8, we only need to regard references r with $r \equiv_{\mathsf{n}} \mathsf{null}$ and show $\overleftarrow{s}_r|_\pi \equiv_{\mathsf{n}}' \mathsf{null}$ for all π with $s_r|_\pi = r$. In all other cases the claim follows from Lemma 3.8. Furthermore, w.l.o.g. we assume that $r \not\equiv \mathsf{null}$. We only need to consider r with $h_r(r) \in \textsc{Instances} \cup \textsc{Arrays}$.

Let π with $s_r|_\pi = r$. We have $r \in [\mathsf{null}]_{\equiv_{\mathsf{n}}}$ because of one of the following cases:

(i) Let $\mathring{r}$ with $s_{\mathring{r}}|_{\mathring{\pi}} = \mathring{r} \equiv r$. We have $t_r(r) \cap t_{\mathring{r}}(\mathring{r}) = \emptyset$. With $s' \sqsubseteq s$ and Definition 3.25(g) we have $\overleftarrow{t}_r(\overleftarrow{s}_r|_\pi) \subseteq t_r(r)$ and $\overleftarrow{t}_{\mathring{r}}(\overleftarrow{s}_{\mathring{r}}|_{\mathring{\pi}}) \subseteq t_{\mathring{r}}(\mathring{r})$. Thus, we also have $\overleftarrow{t}_r(\overleftarrow{s}_r|_\pi) \cap \overleftarrow{t}_{\mathring{r}}(\overleftarrow{s}_{\mathring{r}}|_{\mathring{\pi}}) = \emptyset$ and, hence, $r \equiv_{\mathsf{n}}' \mathsf{null}$. Thus, the claim follows.

(ii) Let $\mathring{\pi}, \mathring{r}$ as in the previous case with $s_{\mathring{r}} = s_r$ and $r \neq \mathring{r}$. We know that $r =^? \mathring{r}$ does not exist. With $s' \sqsubseteq s$ and Definition 3.25(q) we know $\overleftarrow{s}_r|_\pi =^? \overleftarrow{s}_{\mathring{r}}|_{\mathring{\pi}}$ also does not exist. With Definition 3.25(l) we also know that $\overleftarrow{s}_r|_\pi \neq \overleftarrow{s}_{\mathring{r}}|_{\mathring{\pi}}$. Thus, the claim follows.

(iii) Assume we have $s_{r'}|_{\pi'} = r' \underset{\equiv}{\rightarrowtriangle} r = s_r|_\pi$ with $s_r = s_{r'}$. We also may assume that $r \bigvee r'$ does not exist. According to Lemma 3.10 we have $\overleftarrow{s}_{r'}|_{\pi'} \in \overleftarrow{s}_r|_\pi$.

If $\overleftarrow{s}_{r'}|_{\pi'} \rightarrowtriangle \overleftarrow{s}_r|_\pi$, with Definition 3.17(iii) we have $\overleftarrow{s}_{r'}|_{\pi'} \bigvee \overleftarrow{s}_r|_\pi$ or $r \equiv_{\mathsf{n}}' \mathsf{null}$. Then, with Definition 3.25(r) we know $r \equiv_{\mathsf{n}}' \mathsf{null}$, as $r \bigvee r'$ does not exist.

Otherwise, if $\overleftarrow{s}_{r'}|_{\pi'\tau} \equiv' \overleftarrow{s}_r|_\pi$, with Definition 3.18 we may have $\overleftarrow{s}_{r'}|_{\pi'\tau} = \overleftarrow{s}_r|_\pi$ or $\overleftarrow{s}_{r'}|_{\pi'\tau} =^? \overleftarrow{s}_r|_\pi$. As $\pi'\tau \notin \mathrm{SPos}(s_{r'})$, with Definition 3.25(m) we have $r \bigvee r'$. Again, this is a contradiction. Thus, we have $\overleftarrow{s}_r|_\pi \equiv_{\mathsf{n}}' \mathsf{null}$.

(iv) Let $s_{r_1}|_{\pi_1} = r_1 \xrightarrow{\tau_1}\!\!\!\!\!\!\twoheadrightarrow r_2 = s_{r_2}|_{\pi_2}$ and $r_1 \overset{\tau_2}{\twoheadrightarrow} r_2' = s_{r_2'}|_{\pi_2'}$ with $r_2 \equiv r_2'$, $\varepsilon \neq \tau_1$, $\tau_1 \neq \tau_2$, and where τ_1, τ_2 have no corresponding intermediate reference from r_1 in s_{r_1}. With Lemma 3.9 we have $\overleftarrow{s}_{r_1}|_{\pi_1} \overset{\tau_1}{\twoheadrightarrow} \overleftarrow{s}_{r_2}|_{\pi_2}$ and $\overleftarrow{s}_{r_1}|_{\pi_1} \overset{\tau_2}{\twoheadrightarrow} \overleftarrow{s}_{r_2'}|_{\pi_2'}$.

If τ_1, τ_2 have a common intermediate reference from π_1 in $\overleftarrow{s}_{r_1}$, let $\tau_1 = \mathring{\tau}_1\hat{\tau}_1$ with $\mathring{\tau}_1 \neq \varepsilon$ and $\tau_2 = \mathring{\tau}_2\hat{\tau}_2$ with $\mathring{\tau}_2 \neq \varepsilon$ such that $\overleftarrow{s}_{r_1}|_{\pi_1\mathring{\tau}_1} = \overleftarrow{s}_{r_1}|_{\pi_1\mathring{\tau}_2}$ and $\hat{\tau}_1, \hat{\tau}_2$ have no common intermediate reference from $\pi_1\mathring{\tau}_1$ in $\overleftarrow{s}_{r_1}$. With Definition 1.10(l) we also have $s_{r_1}|_{\pi_1\mathring{\tau}_1} = s_{r_1}|_{\pi_1\mathring{\tau}_2}$, or $s_{r_1}|_{\pi_1\mathring{\tau}_1} =^? s_{r_1}|_{\pi_1\mathring{\tau}_2}$, or $\{\pi_1\mathring{\tau}_1, \pi_1\mathring{\tau}_2\} \not\subseteq \mathrm{SP{\small OS}}(s_{r_1})$. In the first case we have a contradiction.

In all other cases, also if no common intermediate reference in $\overleftarrow{s}_{r_1}$ exists as described above, the claim follows with Definition 1.10(n).

(v) Let $r_a \xrightarrow{\tau_a} r$ and $r_b \xrightarrow{\tau_b} r$ with $\tau_a \neq \tau_b$ or $r_a \neq r_b$. Let $s_{r_a}|_{\pi_a} = r_a, s_{r_b}|_{\pi_b} = r_b$, and $s_r|_\pi = r$. We also have $s_{r_a} = s_{r_b}$ and $r_a \mathrel{\bigvee\!\!\!/} r_b$ is missing, furthermore we have $\overline{\pi_a\tau_a}_{s_a} = \pi_a$ and $\overline{\pi_b\tau_b}_{s_b} = \pi_b$.

With Lemma 3.10 we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} \overset{\tau'_a}{-\!\!\in} \overleftarrow{\tilde{s}}_r|_\pi$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}} \overset{\tau'_b}{-\!\!\in} \overleftarrow{\tilde{s}}_r|_\pi$.

First consider that we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} \xrightarrow{\tau'_a} \overleftarrow{\tilde{s}}_r|_\pi$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}} \xrightarrow{\tau'_b} \overleftarrow{\tilde{s}}_r|_\pi$. If $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} \neq \overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}}$ or $\tau'_a \neq \tau'_b$, with Definition 3.17(v) we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} \mathrel{\bigvee\!\!\!/} \overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}}$ or $\overleftarrow{\tilde{s}}_r|_\pi \equiv'_{\mathsf{n}} \mathsf{null}$. In the former case, with Definition 3.25(r) we have $s_{r_a}|_{\pi_a} \mathrel{\bigvee\!\!\!/} s_{r_b}|_{\pi_b}$. Due to this contradiction, we know $\overleftarrow{\tilde{s}}_r|_\pi \equiv'_{\mathsf{n}} \mathsf{null}$. If we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} = \overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}}$ and $\tau'_a = \tau'_b$, with Definition 3.25(m) we have $s_{r_a}|_{\pi_a} \mathrel{\bigvee\!\!\!/} s_{r_b}|_{\pi_b}$. As this results in a conflict, we may disregard this case.

If exactly one of $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} \rightarrow \overleftarrow{\tilde{s}}_r|_\pi$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}} \rightarrow \overleftarrow{\tilde{s}}_r|_\pi$, with Definition 3.17(iii) and Definition 3.25(r) we have $s_{r_a}|_{\pi_a} \mathrel{\bigvee\!\!\!/} s_{r_b}|_{\pi_b}$ or $\overleftarrow{\tilde{s}}_r|_\pi \equiv'_{\mathsf{n}} \mathsf{null}$. If $\{\overleftarrow{s}_{r_a}|_{\pi_a\tau_a}, \overleftarrow{s}_{r_b}|_{\pi_b\tau_b}\} \subseteq [\overleftarrow{\tilde{s}}_r|_\pi]_{\equiv'_{\mathsf{n}}}$, with Definition 3.17(ii) and Definition 3.25(m) we have $s_{r_a}|_{\pi_a} \mathrel{\bigvee\!\!\!/} s_{r_b}|_{\pi_b}$ or $\overleftarrow{\tilde{s}}_r|_\pi \equiv'_{\mathsf{n}} \mathsf{null}$. □

As part of showing $\tilde{s}' \sqsubseteq \tilde{s}$ we first show that a state $\tilde{s}'$ exists.

Lemma 3.12 (Context Concretization exists) Let $s', s, \tilde{s}', \ddot{s}$ as in Definition 3.26. If $\mathrm{cc}(s', \ddot{s}) = \tilde{s}'$, then we also have $\mathrm{cc}(s, \ddot{s}) = \tilde{s}$.

Proof. As $\tilde{s}'$ is a context concretization of s' with $\ddot{s}$, we have $|s'| > 0$. With $|s| = |s'| = n+1$ we then also have $|s| > 0$. We also need to have that pp_n and $\ddot{pp}_0$ are in the same method. With $s' \sqsubseteq s$ and Definition 3.25(a) this claim follows. Furthermore, we must show that there is no class cl with $ic(cl) = \mathrm{NO}$ and $\ddot{ic}(cl) \in \{\mathrm{YES}, \mathrm{RUNNING}\}$, nor $ic(cl) = \mathrm{RUNNING}$ and $\ddot{ic}(cl) = \mathrm{YES}$. With Definition 3.25(c) we have $ic = ic'$. Thus, the claim follows.

Assume we have $\tilde{s}'|_\pi = r$ where r is a return address. Then we know that $\pi \in \{\mathrm{LV}_{i,j}, \mathrm{OS}_{i,j}\}$. We have $\pi \in \mathrm{SP{\small OS}}(\tilde{s})$ and with Definition 3.25(d) and $r = \sigma'(r) = \sigma(r)$

the claim follows. We need to show that the input arguments of s match those of $\ddot{s}$. This is trivial, as with $s' \sqsubseteq s$, $|s'| = |s|$ and Definition 3.25(s) the claim directly follows.

We need to show that for $\equiv_{\mathsf{n}}$ none of the used intersections results in ※. Thus, assume we have $r \equiv_{\mathsf{n}} r'$. Let π, π' be positions with $s_r|_\pi \equiv_{\mathsf{n}} s_{r'}|_{\pi'}$. With Lemma 3.11 we also have $\overleftarrow{s}_r|_\pi \equiv'_{\mathsf{n}} \overleftarrow{s}_{r'}|_{\pi'}$. As the context concretization of s' with $\ddot{s}$ exists, we have $(\overleftarrow{h}_r(\overleftarrow{s}_r|_\pi), \checkmark/\lightning) \Cap^{\checkmark}_{\lightning} (\overleftarrow{h}_{r'}(\overleftarrow{s}_{r'}|_{\pi'}), \checkmark/\lightning) \neq (\text{※}, \checkmark/\lightning)$. With $s' \sqsubseteq s$ we conclude that also $(h_r(s_r|_\pi), \checkmark/\lightning) \Cap^{\checkmark}_{\lightning} (h_{r'}(s_{r'}|_{\pi'}), \checkmark/\lightning) \neq (\text{※}, \checkmark/\lightning)$.

Finally, we also need to show that there is no conflict according to Definition 3.18. For that, assume we have a reference $r \equiv_{\mathsf{n}} \mathsf{null}$ where $r \neq \mathsf{null}$ and $r?$ is missing, or $h_r(r) \in \textsc{Arrays}$, or $h_r(r) = f \in \textsc{Instances}$ with $\mathrm{dom}(f) \neq \emptyset$. As there is no conflict w.r.t. s' and $\ddot{s}$, we know $\overleftarrow{s}_r|_\pi = \mathsf{null}$ or $\overleftarrow{s}_r|_\pi = f' \in \textsc{Instances}$ with $\mathrm{dom}(f') = \emptyset$ and $\overleftarrow{s}_r|_\pi?$ for all π with $s_r|_\pi = r$. With Lemma 3.11 and $s' \sqsubseteq s$ (Definition 3.25(j)) we know that $h_r(r) \notin \textsc{Arrays}$. Furthermore, according to Definition 3.25(p) we have $r?$ and with Definition 3.25(i) we also have $\mathrm{dom}(f) = \emptyset$. Thus, the claim is shown. $\square$

We now show, similar to Lemma 3.6, that the context concretization of a more abstract state does not have a position which does not exist in the context concretization of the more concrete state. In order to be able to conveniently reference positions in a state resulting out of context concretization, we first introduce a new notation.

Definition 3.27 ($[\pi]_{\equiv_{\mathsf{n}}}$) Let $\equiv_{\mathsf{n}}, \tilde{s}$ as in Definition 3.26. For $\pi \in \mathrm{SP{\small OS}}(\tilde{s})$ we define $[\pi]_{\equiv_{\mathsf{n}}} := [r]_{\equiv_{\mathsf{n}}}$ with $\rho([r]_{\equiv_{\mathsf{n}}}) = \tilde{s}|_\pi$.

Lemma 3.13 (State Positions for Context Concretization) Let $\tilde{s}', \tilde{s}$ be defined as in Definition 3.26. Then we have $\mathrm{SP{\small OS}}(\tilde{s}') \supseteq \mathrm{SP{\small OS}}(\tilde{s})$.

The main idea in the proof is that we only disregard available field information for the context concretization if the non-call state indicates that the corresponding object instance may have changed. As such change markers also must exist in more general states (according to the instance definition), such fields are also disregarded in more general states.

An analogous claim holds for arrays and array index information. Array length information is always retained in context concretization (as it is in the intersection process outlined in Section 1.5), as the length of an array instance is immutable.

Proof. We show the claim using a structural induction on π. Let $\pi \in \mathrm{SP{\small OS}}(\tilde{s})$. If $\pi = \mathrm{IA}_{i,\gamma}$ the claim follows from Definition 3.25(s) and Definition 3.23. If $\pi = \mathrm{LV}_{i,j}$ or $\pi = \mathrm{OS}_{i,j}$, the claim follows as we have $|s'| = |s|$ and $pp_i = pp'_i$ for all i. If $\pi = \mathrm{SF}_c$, the claim follows with $ic = ic'$. If $\pi = \mathrm{EXC}$ the claim follows from Definition 3.23.

Now, assume we have $\pi = \pi'\,\tau$ with $|\tau| = 1$. By induction we have $\pi' \in \mathrm{SP{\small OS}}(\tilde{s})$. Let $r \in [\pi']_{\equiv_\mathsf{n}}$ with $s_r|_{\widehat{\pi}'} = r$, $\widehat{\pi}'\,\tau \in \mathrm{SP{\small OS}}(s_r)$ and $s_r|_{\widehat{\pi}'\,\tau} \in [\pi]_{\equiv_\mathsf{n}}$.

If we have $\tau = v$, we have $h_r(r) = f \in \mathrm{I{\small NSTANCES}}$ with $v \in \mathrm{dom}(f) \neq \emptyset$. With $s' \sqsubseteq s$ we then have $\overleftarrow{h}_r(\overleftarrow{s}_r|_{\widehat{\pi}'}) = f' \in \mathrm{I{\small NSTANCES}}$ with $v \in \mathrm{dom}(f) \subseteq \mathrm{dom}(f')$. Thus, $\widehat{\pi}'\,v \in \mathrm{SP{\small OS}}(\overleftarrow{s}_r)$. With Definition 3.18 we have $\tilde{h}'(\tilde{s}'|_{\pi'}) = \tilde{f}' \in \mathrm{I{\small NSTANCES}}$. If $[\overleftarrow{s}_r|_{\mathring{\pi}}]^{\checkmark}_{\equiv'_\mathsf{n}}$, with Definition 3.21 it is rather straight-forward that we also have $v \in \mathrm{dom}(\tilde{f}')$. If $[\overleftarrow{s}_r|_{\mathring{\pi}}]^{\lightning}_{\equiv'_\mathsf{n}}$, w.l.o.g. assume $s_r = s$. Thus, by Definition 3.21 we also have $v \in \mathrm{dom}(\tilde{f}')$. Thus, we have $\pi'\,v \in \mathrm{SP{\small OS}}(\tilde{s}')$ and the claim follows.

The proof for $\tau = i$ is similar, and for $\tau = \mathsf{len}$ the claim directly follows from Definition 3.21 (as the length of an array is immutable). $\square$

Lemma 3.14 Let $\equiv'_\mathsf{n}, \equiv_\mathsf{n}$ as in Definition 3.26. Then if $s_r|_{\pi'} = r \in [\pi]_{\equiv_\mathsf{n}}$ we also have $\overleftarrow{s}_r|_{\pi'} \in [\pi]_{\equiv'_\mathsf{n}}$.

Proof. With Lemma 3.13 we know $\pi \in \mathrm{SP{\small OS}}(\tilde{s}')$, thus $[\pi]_{\equiv'_\mathsf{n}} \neq \emptyset$. It suffices to show that there is $s_r|_{\pi'} = r \in [\pi]_{\equiv_\mathsf{n}}$ with $\overleftarrow{s}_r|_{\pi'} \in [\pi]_{\equiv'_\mathsf{n}}$. Then the claim follows with Lemma 3.11.

We show the claim using a structural induction on π. First we consider $|\pi| = 1$.

- If $\pi \in \{\mathrm{SF}_v, \mathrm{EXC}, \mathrm{OS}_{i,j}, \mathrm{LV}_{i,j}, \mathrm{IA}_{i,\gamma}\}$ for $0 \leq i < |s| = |s'|$, according to Definition 3.23 we have $\tilde{s}|_\pi = \sigma(s|_\pi)$, thus $s|_\pi \equiv_\mathsf{n} s_r|_{\pi'}$. With Lemma 3.11 we also have $s'|_\pi \equiv'_\mathsf{n} \overleftarrow{s}_r|_{\pi'}$. As $\tilde{s}'|_\pi = \sigma'(s'|_\pi)$, we have $s'|_\pi \in [\pi]_{\equiv'_\mathsf{n}}$, thus $\overleftarrow{s}_r|_{\pi'} \in [\pi]_{\equiv'_\mathsf{n}}$.

- If $\pi \in \{\mathrm{OS}_{i,j}, \mathrm{LV}_{i,j}, \mathrm{IA}_{i,\gamma}\}$ with $|s| = |s'| \leq i < |\tilde{s}| = |\tilde{s}'|$, let $\ddot{\pi}$ be the same position but in stack frame $i' = i - |s| + 1$, i.e., $\ddot{\pi} \in \{\mathrm{OS}_{i',j}, \mathrm{LV}_{i',j}, \mathrm{IA}_{i',\gamma}\}$ with $1 \leq i' < |\ddot{s}|$. Then we have $\tilde{s}|_\pi = \sigma(\ddot{s}|_{\ddot{\pi}})$, thus $\ddot{s}|_{\ddot{\pi}} \in [\pi]_{\equiv_\mathsf{n}}$ and $s_r|_{\pi'} \equiv_\mathsf{n} \ddot{s}|_{\ddot{\pi}}$.

  With $s|_{\pi'} \equiv_\mathsf{n} \ddot{s}|_{\ddot{\pi}}$ and Lemma 3.11 we also have $\overleftarrow{\ddot{s}}|_{\pi'} \equiv'_\mathsf{n} \ddot{s}|_{\ddot{\pi}}$. As $\tilde{s}'|_\pi = \sigma'(\ddot{s}|_{\ddot{\pi}})$, with $\ddot{s}|_{\ddot{\pi}} \in [\pi]_{\equiv'_\mathsf{n}}$ we then also have $\overleftarrow{\ddot{s}}|_{\pi'} \in [\pi]_{\equiv'_\mathsf{n}}$.

Now we consider $\pi = \pi_1\tau$ with $|\tau| = 1$. By induction we know that for all $s_{r_p}|_{\pi_p} = r_p \in [\pi_1]_{\equiv_\mathsf{n}}$ we have $\overleftarrow{s}_{r_p}|_{\pi_p} \in [\pi_1]_{\equiv'_\mathsf{n}}$.

First assume $\tau = v \in \mathrm{F{\small IELD}ID{\small S}}$. W.l.o.g. pick r_p such that $h_{r_p}(r_p) = f \in \mathrm{I{\small NSTANCES}}$ with $v \in \mathrm{dom}(f)$, and $f(v) = s_{r_p}|_{\pi_p\,v} \in [\pi]_{\equiv_\mathsf{n}}$. Also, if $[\overleftarrow{s}_{r_p}]^{\lightning}_{\equiv_\mathsf{n}}$, we pick r_p such that $s_{r_p} =$

s. Similar to the proof of Lemma 3.13, we have $\overleftarrow{s_{r_p}}|_{\pi_p v} \in [\pi]_{\equiv'_{\mathsf{n}}}$. With $s_{r_p}|_{\pi_p v} \in [\pi]_{\equiv_{\mathsf{n}}}$ and $\overleftarrow{s_{r_p}}|_{\pi_p v} \in [\pi]_{\equiv'_{\mathsf{n}}}$ the claim follows.

As in the proof of Lemma 3.13, the proof for $v = i$ and $v = \mathsf{len}$ is analogous (or even simpler). □

In the following pages we will present more auxialliary lemmas which are used in the upcoming proof of Theorem 3.7.

Preparation for proof step (l)

We first present lemmas which are used to show the proof step corresponding to Definition 3.25(l). Here, for a specific situation we need to have two references in the same state. For this we make use of the input arguments and show that these guarantee existence of such references.

The first lemma shows that for equivalent references this equivalence results out of an input argument if the references are in different states.

Lemma 3.15 Let $s, \ddot{s}, \equiv_{\mathsf{n}}$ as in Definition 3.26.
Assume we have $s_r|_{\pi_1} = r \equiv_{\mathsf{n}} r' = s_2|_{\pi_2}$ with $h(r) \in \textsc{Instances} \cup \textsc{Arrays}$.

(i) If $s_r = s$ and $s_{r'} = \ddot{s}$, we have input arguments $(\lambda, \gamma, \checkmark/\lightning) \in ia_n$ and $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\gamma = \ddot{\gamma}$, $s|_{\mathrm{IA}_{n,\gamma}} \equiv_{\mathsf{n}} r$, and $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r'$.

(ii) Analogously, if $s_r = \ddot{s}$ and $s_{r'} = s$, we have input arguments $(\lambda, \gamma, \checkmark/\lightning) \in ia_n$ and $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\gamma = \ddot{\gamma}$, $s|_{\mathrm{IA}_{n,\gamma}} \equiv_{\mathsf{n}} r'$, and $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r$.

(iii) If $s_r = s_{r'} = \ddot{s}$ and we have an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r$, then we also have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = r'$.

(iv) Analogously, if $s_r = s_{r'} = \ddot{s}$ and we have an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r'$, then we also have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = r$.

Proof. We only show proofs for the first and third item, as the proofs for the other items are analogous (when swapping the roles of r and r'). We have $r \equiv_{\mathsf{n}} r'$ and show the claim using an induction.

- First, consider that we have $s_r = s$ and $s_{r'} = \ddot{s}$.

We know $h(r) \in \textsc{Instances} \cup \textsc{Arrays}$, thus $r \neq r'$. If there is an input argument $(\lambda, \gamma, \checkmark/\lightning) \in ia_n$ with $\lambda = r$ and $\gamma(\ddot{s}) = r'$, we have $r = \lambda \equiv_{\mathsf{n}} \gamma(\ddot{s}) = r'$ as in Definition 3.12(i). With Definition 3.22(vi) we also know that there is an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\gamma = \ddot{\gamma}$. Then we have $s|_{\mathrm{IA}_{n,\gamma}} = r$ and $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r'$. Thus, the claim is shown.

Now assume $r \equiv_{\mathsf{n}} r'$ because we have $s|_{\pi_1'} \equiv_{\mathsf{n}} \ddot{s}|_{\pi_2'}$ as in Definition 3.12(ii) where $\pi_1 = \pi_1'\, v$, $\pi_2 = \pi_2'\, v$, $h(s|_{\pi_1'}) = f_1 \in \textsc{Instances}$, and $\ddot{h}(\ddot{s}|_{\pi_2'}) = f_2 \in \textsc{Instances}$ with $v \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2)$ and $s|_{\pi_1'}\checkmark$. By induction we know there is an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = \ddot{s}|_{\pi_2'}$. Thus, we also have $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}\, v} = r'$. With Definition 3.6(iii) we then also have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = r'$. With Definition 3.22(v) we have an input argument $(\lambda', \gamma', \checkmark/\lightning) \in ia_n$ with $\ddot{\gamma}' = \gamma'$ and $s|_{\mathrm{IA}_{n,\gamma'}} \equiv_{\mathsf{n}} \ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} \equiv_{\mathsf{n}} r' \equiv_{\mathsf{n}} r$. The proof for Definition 3.12(iii) is analogous.

If we have $r \equiv_{\mathsf{n}} r_m$ and $r_m \equiv_{\mathsf{n}} r'$, the proof trivially follows by transitivity of $\equiv_{\mathsf{n}}$ if $s_{r_m} = s$. If we have $r \equiv_{\mathsf{n}} r_m \equiv_{\mathsf{n}} r'$ with $s_{r_m} = s_{r'} = \ddot{s}$, by induction we know there are input arguments $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ and $(\lambda, \gamma, \checkmark/\lightning) \in ia_n$ with $\ddot{\gamma} = \gamma$, $s|_{\mathrm{IA}_{n,\gamma}} \equiv_{\mathsf{n}} r$, and $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r_m$. Furthermore, by induction we know there also is an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = r'$. Thus, the claim is shown.

- Now consider that we have $s_r = s_{r'} = \ddot{s}$ where an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ exists with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = r$.

  If $r = r'$, the claim immediately follows. We cannot have $r \equiv_{\mathsf{n}} r'$ with Definition 3.12(i), as $s_r = s_{r'} = \ddot{s}$. Thus, consider that we have $r \equiv_{\mathsf{n}} r'$ with Definition 3.12(ii) where we have $\ddot{s}|_{\pi_1'} \equiv_{\mathsf{n}} \ddot{s}|_{\pi_2'}$ with $\pi_1 = \pi_1'\, v$, $\pi_2 = \pi_2'\, v$, $\ddot{h}(\ddot{s}|_{\pi_1'}) = f_1 \in \textsc{Instances}$, $\ddot{h}(\ddot{s}|_{\pi_2'}) = f_2 \in \textsc{Instances}$, and $v \in \mathrm{dom}(f_1) \cap \mathrm{dom}(f_2)$. With Definition 3.6(vi) we have $\ddot{s}|_{\pi_3} \rightsquigarrow r$ or $r \rightsquigarrow \ddot{s}|_{\pi_4} \wedge \ddot{s}|_{\pi_3} \rightsquigarrow \ddot{s}|_{\pi_4}$ with $\pi_3 \in \{\mathrm{SF}_v, \mathrm{LV}_{0,j}\}$. In the former case, we have $\ddot{s}|_{\pi_3} \rightsquigarrow r$ and $\ddot{s}|_{\pi_1'} \rightsquigarrow r$. Thus, with Definition 3.6(iv) we have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = \ddot{s}|_{\pi_1'}$. In the latter case, we also have $\ddot{s}|_{\pi_1'} \rightsquigarrow \ddot{s}|_{\pi_4}$. Thus, with Definition 3.6(iv) we also have $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning)$ as above. By induction we have an input argument for $\ddot{s}|_{\pi_2'}$ and, with Definition 3.6(iii), the claim follows. The proof for Definition 3.12(iii) is analogous.

  If we have $r \equiv_{\mathsf{n}} r_m \equiv_{\mathsf{n}} r'$, we either have $s_{r_m} = \ddot{s}$ or $s_{r_m} = s$. In both cases, the claim follows by induction. $\square$

In the following lemma we make use of the input arguments to guarantee that for a specific situation we always have two references which are in the same state.

Lemma 3.16 Let $\tilde{s}', \tilde{s}, \equiv'_{\mathsf{n}}, \equiv_{\mathsf{n}}$ as in Definition 3.26.

For all positions $\{\pi, \pi'\} \subseteq \mathrm{SP{\small OS}}(\tilde{s})$ with $\tilde{s}'|_\pi \equiv'_{\mathsf{n}} \tilde{s}'|_{\pi'}$ and $\tilde{h}'(\tilde{s}'|_\pi) \in \mathrm{I{\small NSTANCES}} \cup \mathrm{A{\small RRAYS}}$ there are r, r' with $r \in [\pi]_{\equiv_{\mathsf{n}}}$, $r' \in [\pi']_{\equiv_{\mathsf{n}}}$, and $s_r = s_{r'}$.

Proof. Let $r = s_r|_{\pi_1} \in [\pi]_{\equiv_{\mathsf{n}}}$ and $r' = s_{r'}|_{\pi_2} \in [\pi']_{\equiv_{\mathsf{n}}}$. If $s_r = s_{r'}$, the claim is shown. Thus, w.l.o.g. assume we have $s_r = s$, and $s_{r'} = \ddot{s}$. Then the claim follows with Lemma 3.15. $\square$

Preparation for proof step (o)

For the proof step for Definition 3.25(o) we need to reason about abstract predecessors of certain references. Thus, in the following lemmas we show some properties for predecessors.

Lemma 3.17 Let $\tilde{s}', \tilde{s}, \equiv'_{\mathsf{n}}, \equiv_{\mathsf{n}}$ as in Definition 3.26. Let $\pi \in \mathrm{SP{\small OS}}(\tilde{s}')$. Let $\overset{\leftarrow}{s}_r|_{\pi'} = r \xrightarrow{\tau} [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$. If $[\overline{\pi}_{\tilde{s}}]^{\not\downarrow}_{\equiv'_{\mathsf{n}}}$, let $s_r = s$.

Then we have $\overset{\leftarrow}{s}_r|_{\pi'} \xrightarrow{\tau\tau'} [\pi]_{\equiv'_{\mathsf{n}}}$ where $\pi = \overline{\pi}_{\tilde{s}}\tau'$.

Proof. If $\overline{\pi}_{\tilde{s}} = \pi$, the claim is trivially shown. Thus, assume $\pi = \overline{\pi}_{\tilde{s}}\tau'$ with $\tau' \neq \varepsilon$. As $\overset{\leftarrow}{s}_r|_{\pi'} \xrightarrow{\tau} [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$, we know $\overline{\pi'\tau}_{\overset{\leftarrow}{s}_r} = \pi'$. With $\pi \in \mathrm{SP{\small OS}}(\tilde{s}')$ the claim follows. $\square$

Lemma 3.18 Let $\tilde{s}', \tilde{s}, \equiv'_{\mathsf{n}}, \equiv_{\mathsf{n}}$ as in Definition 3.26. Let $\pi \in \mathrm{SP{\small OS}}(\tilde{s}')$. Let $\overset{\leftarrow}{s}_r|_{\pi'} = r \in [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$ where $\pi = \overline{\pi}_{\tilde{s}}\tau$. If $[\overline{\pi}_{\tilde{s}}]^{\not\downarrow}_{\equiv'_{\mathsf{n}}}$, let $s_r = s$.

Then we have $\overset{\leftarrow}{s}_r|_{\overline{\pi'\tau}} \overset{\tau'}{-\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$ with $\pi'\tau = \overline{\pi'\tau}_{\overset{\leftarrow}{s}_r}\tau'$.

Proof. If $\overline{\pi}_{\tilde{s}} = \pi$, the claim is trivially shown. Thus, assume $\pi = \overline{\pi}_{\tilde{s}}\tau$ with $\tau \neq \varepsilon$. If $\overset{\leftarrow}{s}_r = \ddot{s}$ we know $\overline{\pi'\tau}_{\ddot{s}} = \pi'$ or $[\overline{\pi}_{\tilde{s}}]^{\not\downarrow}_{\equiv_{\mathsf{n}}}$. In the latter case we also have $\overset{\leftarrow}{s}_r = s$, thus for $\overset{\leftarrow}{s}_r = \ddot{s}$ we know $\overline{\pi'\tau}_{\ddot{s}} = \pi'$ and $[\overline{\pi}_{\tilde{s}}]^{\checkmark}_{\equiv'_{\mathsf{n}}}$. Thus, if $\overset{\leftarrow}{s}_r = \ddot{s}$ we have $\ddot{s}|_{\overline{\pi'\tau}} \xrightarrow{\tau} [\pi]_{\equiv'_{\mathsf{n}}}$. Otherwise, if $s_r = s$ we may have $\overline{\pi'\tau}_s \neq \pi'$. If $\overline{\pi'\tau}_s = \pi'\tau$ we have $s'_r|_{\pi'\tau} \in [\pi]_{\equiv'_{\mathsf{n}}}$. Otherwise we have $s'_r|_{\overline{\pi'\tau}} \xrightarrow{\tau'} [\pi]_{\equiv'_{\mathsf{n}}}$ with $\overline{\pi'\tau}_s\tau' = \pi'\tau$. $\square$

Lemma 3.19 Let $\tilde{s}', \tilde{s}$ as in Definition 3.26. Let $\pi \in \textsc{SPos}(\tilde{s}')$ and let τ with $\pi = \overline{\pi}_{\tilde{s}}\tau$. Let $s_r|_{\pi_a} = r \overset{\tau_a}{-\!\!\!\in} [\overline{\pi}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$ where $s_r = s$ if $[\overline{\pi}_{\tilde{s}}]^{\lightning}_{\equiv_{\mathsf{n}}}$.

Then we have $\overleftarrow{s}_r|_{\overline{\pi_a\tau_a\tau}} \overset{\tau'}{-\!\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$ with $\overline{\pi_a\tau_a\tau}_{\overleftarrow{s}_r}\tau' = \pi_a\tau_a\tau$.

Proof. If $s_r|_{\pi_a} \in [\overline{\pi}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$, with Lemma 3.14 we have $\overleftarrow{s}_r|_{\pi_a} \in [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$. According to Lemma 3.18 we have $\overleftarrow{s}_r|_{\overline{\pi_a\tau}} \overset{\tau'}{-\!\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$ with $\pi_a\tau = \overline{\pi_a\tau}_{\overleftarrow{s}_r}\tau'$. If $s_r|_{\pi_a} \overset{\tau_a}{\rightarrowtriangle} [\overline{\pi}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$, with Lemma 3.10 we have $\overleftarrow{s}_r|_{\overline{\pi_a\tau_a}} \overset{\tau'_a}{-\!\!\!\in} [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$ with $\overline{\pi_a\tau_a}_{\overleftarrow{s}_r}\tau'_a = \pi_a\tau_a$. If $\overleftarrow{s}_r|_{\overline{\pi_a\tau_a}} \overset{\tau'_a}{\rightarrowtriangle} [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$, with Lemma 3.17 we also have $\overleftarrow{s}_r|_{\overline{\pi_a\tau_a}} \overset{\tau'_a\tau}{\longrightarrowtriangle} [\pi]_{\equiv'_{\mathsf{n}}}$. If $\overleftarrow{s}_r|_{\pi_a\tau_a} \in [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$, with Lemma 3.18 we have $\overleftarrow{s}_r|_{\overline{\pi_a\tau_a\tau}} \overset{\tau'}{-\!\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$ with $\overline{\pi_a\tau_a\tau}_{\overleftarrow{s}_r}\tau' = \pi_a\tau_a\tau$. □

Preparation for proof step (m)

Finally, as a last preparation step, we deal with Definition 3.25(m). For this, we directly show that the necessary heap predicates exist. For this we make use of the lemmas previously presented, and also show that (similar to the case of Definition 3.25(l)) for a specific situation we always have references in the same state as guaranteed by the input arguments.

Lemma 3.20 Let $\tilde{s}', \tilde{s}$ as in Definition 3.26. Assume we have $\tilde{s}'|_{\pi} = \tilde{s}'|_{\pi'}$ or $\tilde{s}'|_{\pi} =^{?} \tilde{s}'|_{\pi'}$ with $\pi \notin \textsc{SPos}(\tilde{s})$, $\tilde{h}'(\tilde{s}'|_{\pi}) \in \textsc{Instances} \cup \textsc{Arrays}$, and $\pi \neq \pi'$. Furthermore we have $\tilde{s}|_{\overline{\pi}} \neq \tilde{s}|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. $\tilde{s}$. Let $s_{r_a}|_{\pi_a} = r_a -\!\!\!\in [\overline{\pi}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$ and $s_{r_b}|_{\pi_b} = r_b -\!\!\!\in [\overline{\pi'}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$.

Then, if $s_{r_a} = s_{r_b}$, we have $r_a \curlyvee r_b$. If $[\overline{\pi}_{\tilde{s}}]^{\lightning}_{\equiv_{\mathsf{n}}}$ or $[\overline{\pi'}_{\tilde{s}}]^{\lightning}_{\equiv_{\mathsf{n}}}$ we only show this for $s_{r_a} = s_{r_b} = s$.

Proof. Let $\pi = \overline{\pi}_{\tilde{s}}\tau$ and let $\pi' = \overline{\pi'}_{\tilde{s}}\tau'$. With Lemma 3.19 we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a\tau}} -\!\!\!\in [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b\tau'}} -\!\!\!\in [\pi']_{\equiv'_{\mathsf{n}}}$. If $\overleftarrow{s}_{r_a}|_{\pi_a\tau_a\tau} \in [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s}_{r_b}|_{\pi_b\tau_b\tau'} \in [\pi']_{\equiv'_{\mathsf{n}}}$, with Definition 3.17(ii) and Definition 3.23(b) we have $\overleftarrow{s}_{r_a}|_{\pi_a\tau_a\tau} = \overleftarrow{s}_{r_b}|_{\pi_b\tau_b\tau'}$ or $\overleftarrow{s}_{r_a}|_{\pi_a\tau_a\tau} =^{?} \overleftarrow{s}_{r_b}|_{\pi_b\tau_b\tau'}$. Thus, with Definition 3.25(m) we may have $s_{r_a}|_{\pi_a} \curlyvee s_{r_b}|_{\pi_b}$. Otherwise, we have $r_a = r_b$ and $\pi_a\tau_a\tau, \pi_b\tau_b\tau'$ have the same suffix w.r.t. $s_{r_a} = s_{r_b}$. From this we can conclude that $\tau_a = \tau_b$. As $[\overline{\pi}_{\tilde{s}}]_{\equiv_{\mathsf{n}}} \neq [\overline{\pi'}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$ or $\tau \neq \tau'$, we have $r_a \neq r_b$. Thus, we know $r_a \curlyvee r_b$.

Now consider that we have $\overleftarrow{s}_{r_a}|_{\pi_a\tau_a\tau} \in [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b\tau'}} \rightarrowtriangle [\pi']_{\equiv'_{\mathsf{n}}}$ If $[\pi]_{\equiv_{\mathsf{n}}} = [\pi']_{\equiv_{\mathsf{n}}}$ we also have $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b\tau'}} \rightarrowtriangle \overleftarrow{s}_{r_a}|_{\pi_a\tau_a\tau}$. Thus, with Definition 3.17(iii) and Definition 3.25(r) we have $r_a \curlyvee r_b$. Otherwise, we have $[\pi]_{\equiv_\mathsf{n}} \neq [\pi']_{\equiv_\mathsf{n}}$. Then, with Definition 3.23(b) we have $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b\tau'}} \curlyvee \overleftarrow{s}_{r_a}|_{\pi_a\tau_a\tau}$. Thus, with Definition 3.25(r) we have $r_a \curlyvee r_b$. The case where we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a\tau}} \rightarrowtriangle [\pi]_{\equiv'_\mathsf{n}}$ and $\overleftarrow{s}_{r_b}|_{\pi_b\tau_b\tau'} \in [\pi']_{\equiv'_\mathsf{n}}$ is analogous.

Lastly, consider the case where we have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a\tau}} \rightarrowtriangle [\pi]_{\equiv'_\mathsf{n}}$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b\tau'}} \rightarrowtriangle [\pi']_{\equiv'_\mathsf{n}}$. Then, with Definition 3.23(b), Definition 3.17(v), and Definition 3.25(m,r) we have $r_a \curlyvee r_b$. $\square$

Lemma 3.21 Let $s, \ddot{s}, \equiv_\mathsf{n}$ as in Definition 3.26. If we have $s|_\pi \mathrel{-\!\!\in} \ddot{s}|_{\pi'}$ with $\ddot{h}(\ddot{s}|_{\pi'}) \in$ Instances $\cup$ Arrays, then there is an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = \ddot{s}|_{\pi'}$. We also have $s|_{\mathrm{IA}_{n,\ddot{\gamma}}} \equiv_\mathsf{n} \ddot{s}|_{\pi'}$.

Proof. Let $s|_\pi \overset{\tau}{\mathrel{-\!\!\in}} \ddot{s}|_{\pi'}$. We show the claim using an induction on τ. If $\tau = \varepsilon$, we have $s|_\pi \equiv_\mathsf{n} \ddot{s}|_{\pi'}$. Thus, with Lemma 3.15 the claim follows.

Now let $\tau = \tau'\tau''$ with $|\tau''| = 1$. Thus, we have $s|_\pi \overset{\tau}{\rightarrowtriangle} \ddot{s}|_{\pi'}$. With Definition 3.16 we know $\overline{\pi\tau'\tau''}_s = \pi$ and $s|_\pi \overset{\tau}{\twoheadrightarrow} \ddot{s}|_{\pi'}$.

If $\tau' = \varepsilon$, with Definition 3.15 we have $s|_\pi \equiv_\mathsf{n} s_x|_{\pi_x}$ for some $s_x|_{\pi_x}$ with $s_x|_{\pi_x\tau''} \equiv_\mathsf{n} \ddot{s}|_{\pi'}$. If $s_{r_x} = s$, we have $s|_{\pi_x\tau''} \equiv_\mathsf{n} \ddot{s}|_{\pi'}$. Then, with Lemma 3.15 the claim follows. If $s_{r_x} = \ddot{s}$, with Lemma 3.15 we have an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = \ddot{s}|_{\pi_x}$. With Definition 3.6(iii) we also have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = \ddot{s}|_{\pi_x\tau''}$. Then the claim follows with Lemma 3.15.

If $\tau' \neq \varepsilon$, we have $s|_\pi \overset{\tau'}{\mathrel{-\!\!\in}} s_x|_{\pi_x}$ and $s_x|_{\pi_x\tau''} \equiv_\mathsf{n} \ddot{s}|_{\pi'}$ for some $s_x|_{\pi_x}$. If $s_x = \ddot{s}$, by induction we have an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}}} = \ddot{s}|_{\pi_x}$. With Definition 3.6(iii) we also have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\lightning) \in \ddot{ia}_0$ with $\ddot{s}|_{\mathrm{IA}_{0,\ddot{\gamma}'}} = \ddot{s}|_{\pi_x\tau''}$. If $s_x = s$, we have $s|_{\pi_x\tau''} \equiv_\mathsf{n} \ddot{s}|_{\pi'}$. In both cases the claim follows with Lemma 3.15. $\square$

Lemma 3.22 Let $\tilde{s}', \tilde{s}$ as in Definition 3.26. Furthermore, assume we have $\tilde{s}'|_\pi = \tilde{s}'|_{\pi'}$ or $\tilde{s}'|_\pi =^? \tilde{s}'|_{\pi'}$ with $\{\pi, \pi'\} \not\subseteq \textsc{SPos}(\tilde{s})$, $\tilde{h}'(\tilde{s}'|_\pi) \in$ Instances $\cup$ Arrays, and $\pi \neq \pi'$.

Then there are $s_{r_1}|_{\pi_1} = r_1 \in [\overline{\pi}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ and $s_{r_2}|_{\pi_2} = r_2 \in [\overline{\pi'}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ with $s_{r_1} = s_{r_2}$.

Proof. Let $s_{r_a}|_{\pi_a} = r_a \in [\overline{\pi}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ and $s_{r_b}|_{\pi_b} = r_b \in [\overline{\pi'}_{\tilde{s}}]_{\equiv_\mathsf{n}}$. If there are such r_a, r_b with $s_{r_b} = s_{r_b}$, the claim is shown.

W.l.o.g. assume for all $s_{r_a}|_{\pi_a} \in [\overline{\pi}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ we have $s_{r_a} = s$, and for all $s_{r_b}|_{\pi_b} \in [\overline{\pi'}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ we have $s_{r_b} = \ddot{s}$.

We have $\pi' \in \text{SPos}(\tilde{s}')$. If $[\overline{\pi'}_{\tilde{s}}]^{\not\downarrow}_{\equiv_\mathsf{n}}$, we know there is a reference $s_{r_c}|_{\pi_c} \in [\overline{\pi'}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ with $s_{r_c}|_{\pi_c}\not\downarrow$. With Definition 3.8 we know $s_{r_c} = s$. Otherwise, if $[\overline{\pi'}_{\tilde{s}}]^{\checkmark}_{\equiv_\mathsf{n}}$, we have $s_{r_c}|_{\pi_c} \in [\overline{\pi'}_{\tilde{s}}]_{\equiv'_\mathsf{n}}$ with $s_{r_c}|_{\pi_c\tau} \in [\overline{\pi'}_{\tilde{s}}\tau]_{\equiv'_\mathsf{n}}$ and $\overline{\pi'}_{\tilde{s}}\tau \trianglelefteq \pi'$ where, if $\pi' \notin \text{SPos}(\tilde{s})$, we have $\tau \neq \varepsilon$. If $s_{r_c} = s'$, with Lemma 3.15 we have an input argument $(\lambda', \gamma', \checkmark/\not\downarrow) \in ia'_n$ with $s'|_{\text{IA}_{n,\gamma'}} \equiv'_\mathsf{n} \ddot{s}|_{\pi_b}$. With Definition 3.25(s) we then also have $(\lambda, \gamma, \checkmark/\not\downarrow) \in ia_n$ with $\gamma = \gamma'$ and $s|_{\text{IA}_{n,\gamma}} \equiv_\mathsf{n} \ddot{s}|_{\pi_b}$.

If $s_{r_c} = \ddot{s}$ and $\ddot{s}|_{\pi_c\tau} \in [\overline{\pi'}_{\tilde{s}}\tau]_{\equiv'_\mathsf{n}}$ for τ as above, we have a contradiction if $\pi' \notin \text{SPos}(\tilde{s})$. Thus we only need to consider the case that $\pi \notin \text{SPos}(\tilde{s})$ and $\pi' \in \text{SPos}(\tilde{s})$, thus also $\overline{\pi'}_{\tilde{s}} = \pi'$. According to Lemma 3.18 we have $s'|_{\overline{\pi_a\tau'}} \in [\pi]_{\equiv'_\mathsf{n}}$ for $\overline{\pi}_{\tilde{s}}\tau' = \pi$.

If $[\pi]_{\equiv'_\mathsf{n}} = [\pi']_{\equiv'_\mathsf{n}}$, we know $\ddot{s}|_{\pi_b} \in [\pi]_{\equiv'_\mathsf{n}}$. Thus, we also have $s'|_{\overline{\pi_a\tau'}} \in \ddot{s}|_{\pi_b}$. With Lemma 3.21 we then also have an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\not\downarrow) \in \ddot{ia}_0$ with $\ddot{s}|_{\text{IA}_{0,\gamma}} = \ddot{s}|_{\pi_b}$. With Definition 3.22(v) we have $s|_{\text{IA}_{n,\ddot{\gamma}}} \equiv_\mathsf{n} \ddot{s}|_{\pi_b} \equiv_\mathsf{n} [\overline{\pi'}_{\tilde{s}}]_{\equiv_\mathsf{n}}$.

If $\tilde{s}'|_\pi =^? \tilde{s}'|_{\pi'}$, with Definition 3.23(b) we know $\ddot{s}|_{\pi_d} \in [\pi]_{\equiv'_\mathsf{n}}$ for some π_d, as only $\ddot{s}$ contains references equivalent to $[\pi']_{\equiv_\mathsf{n}}$. Thus, with $s'|_{\overline{\pi_a\tau'}} \in \ddot{s}|_{\pi_d}$ and Lemma 3.21 we also have an input argument $(\ddot{\lambda}, \ddot{\gamma}, \checkmark/\not\downarrow) \in \ddot{ia}_0$ with $\ddot{s}|_{\text{IA}_{0,\ddot{\gamma}}} = \ddot{s}|_{\pi_d}$. As we have $\ddot{s}|_{\pi_d} =^? \ddot{s}|_{\pi_b}$, with Definition 3.6(v) we then also have an input argument $(\ddot{\lambda}', \ddot{\gamma}', \checkmark/\not\downarrow) \in \ddot{ia}_0$ with $\ddot{s}|_{\text{IA}_{0,\ddot{\gamma}'}} = \ddot{s}|_{\pi_b}$. Then, with Definition 3.22(v) we have $s|_{\text{IA}_{n,\ddot{\gamma}'}} \equiv_\mathsf{n} \ddot{s}|_{\pi_b} \in [\pi']_{\equiv_\mathsf{n}}$.

Thus, the claim is shown. $\square$

Lemma 3.23 Let $\tilde{s}', \tilde{s}$ as in Definition 3.26. Assume we have $\tilde{s}'|_\pi = \tilde{s}'|_{\pi'}$ or $\tilde{s}'|_\pi =^? \tilde{s}'|_{\pi'}$ with $\pi \notin \text{SPos}(\tilde{s})$, $\tilde{h}'(\tilde{s}'|_\pi) \in \text{Instances} \cup \text{Arrays}$, and $\pi \neq \pi'$. Furthermore we have $\tilde{s}|_{\overline{\pi}} \neq \tilde{s}|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. $\tilde{s}$.

Then we have $\tilde{s}|_{\overline{\pi}_{\tilde{s}}} \searrow\!\!\!\swarrow \tilde{s}|_{\overline{\pi'}_{\tilde{s}}}$.

Proof. Let $s_{r_a}|_{\pi_a} = r_a \in [\overline{\pi}_{\tilde{s}}]_{\equiv_\mathsf{n}}$ and $s_{r_b}|_{\pi_b} = r_b \in [\overline{\pi'}_{\tilde{s}}]_{\equiv_\mathsf{n}}$. If $s_{r_a} = s_{r_b}$, with Lemma 3.20 we have $r_a \searrow\!\!\!\swarrow r_b$. According to Lemma 3.22 we know that such r_a, r_b exist. Thus, with Definition 3.23(c,d) the claim follows. $\square$

Proof

Now, after defining and proving correct 17 auxiliary lemmas, we finally show that Theorem 3.7 holds.

Proof. *(of Theorem 3.7 on page 147)* We make use of the symbols as in Definition 3.26.

With Lemma 3.12 we have a context concretization $\tilde{s}$ of s with $\ddot{s}$. Thus, we need to show $\tilde{s}' \sqsubseteq \tilde{s}$. As we have $|s'| = |s|$, we also have $|\tilde{s}'| = |\tilde{s}|$.

We need to show Definition 3.25(a–s).

(a) With $s' \sqsubseteq s$ we have $\tilde{pp}'_i = pp'_i = \tilde{pp}_i = pp_i$ for all $0 \leq i \leq n$ and $\tilde{pp}'_i = \ddot{pp}_{i-n} = \tilde{pp}_i$ for all $n < i \leq n + m$.

(b) If $\tilde{s}'|_{\text{EXC}} = \perp$, we also have $s'|_{\text{EXC}} = \perp$. With Definition 3.25(b) we also have $s|_{\text{EXC}} = \perp$ and $\tilde{s}|_{\text{EXC}} = \perp$. Similarly, if $\tilde{s}'|_{\text{EXC}} \neq \perp$, we also have $\tilde{s}|_{\text{EXC}} \neq \perp$.

(c) We have $\widetilde{ic}' = ic' = ic = \widetilde{ic}$.

Let $\pi \in \text{SP}\textsc{os}(\tilde{s})$. According to Lemma 3.14 for each $r \in [\pi]_{\equiv_\mathsf{n}}$ with $s_r|_{\pi'} = r$ we have $\overleftarrow{s_r}|_{\pi'} \in [\pi]_{\equiv'_\mathsf{n}}$.

(d) $\tilde{s}'|_\pi$ is a return address. Then, with Definition 3.25(d) we have $\tilde{s}'|_\pi = \tilde{s}|_\pi$.

(e) We have $\tilde{h}'(\tilde{s}'|_\pi) = \widetilde{V}' \in \textsc{Floats}$. Let $\tilde{h}(\tilde{s}|_\pi) = \widetilde{V}$.

Thus, for each r as above we have $\overleftarrow{h_r}(\overleftarrow{s_r}|_{\pi'}) \in \{\widetilde{V}', \perp\}$. With Definition 3.25(e) we also have $h_r(r) \in \{\widetilde{V}', \perp\}$. Thus, we have $\widetilde{V} \in \{\widetilde{V}', \perp\}$.

(f) We have $\tilde{h}'(\tilde{s}'|_\pi) = \widetilde{V}' \in \textsc{Integers}$. Let $\tilde{h}(\tilde{s}|_\pi) = \widetilde{V}$.

For each r as above we have $\widetilde{V}' \subseteq \overleftarrow{h_r}(\overleftarrow{s_r}|_{\pi'})$. With Definition 3.25(f) we also have $\widetilde{V}' \subseteq h_r(r)$. Thus, we have $\widetilde{V}' \subseteq \widetilde{V}$.

(g) We have $\tilde{t}'(\tilde{s}'|_\pi) = \widetilde{V}' \in \textsc{Types}$. Let $\tilde{t}(\tilde{s}|_\pi) = \widetilde{V}$.

For each r as above we have $\widetilde{V}' \subseteq \overleftarrow{t_r}(\overleftarrow{s_r}|_{\pi'})$. With Definition 3.25(g) we also have $\widetilde{V}' \subseteq t_r(r)$. Thus, we have $\widetilde{V}' \subseteq \widetilde{V}$.

(h) We have $\tilde{s}'|_\pi = \mathsf{null}$. Let $\tilde{s}|_\pi = \tilde{r}$.

According Definition 3.18 for each r as above we have $\overleftarrow{s_r}|_{\pi'} = \mathsf{null}$ or $\overleftarrow{h_r}(\overleftarrow{s_r}|_{\pi'}) = f \in \textsc{Instances}$ with $\text{dom}(f) = \emptyset$ and $\overleftarrow{s_r}|_{\pi'}?$. With Definition 3.25(h,i,p) we also have $s_r|_r = \mathsf{null}$ or $h_r(r) = f \in \textsc{Instances}$ with $\text{dom}(f) = \emptyset$ and $r?$. Thus, with Definition 3.23(a) we have $\tilde{r} = \mathsf{null}$ or $\tilde{h}(\tilde{r}) = \tilde{f} \in \textsc{Instances}$ with $\text{dom}(\tilde{f}) = \emptyset$ and $\tilde{r}?$.

(i) We have $\tilde{h}'(\tilde{s}'|_\pi) = \tilde{f}' \in \textsc{Instances}$ and $\pi \in \text{SP}\textsc{os}(\tilde{s})$.

According to Definition 3.21, for each r as above we have $\overleftarrow{h_r}(\overleftarrow{s_r}|_{\pi'}) \in \textsc{Instances}$. Furthermore, we have $\text{dom}(\tilde{f}') \supseteq \text{dom}(\overleftarrow{h_r}(\overleftarrow{s_r}|_{\pi'}))$ if $\overleftarrow{s_r} = s'$ or $[\pi]^{\checkmark}_{\equiv'_\mathsf{n}}$. With Definition 3.25(i) we also have $\text{dom}(\tilde{f}') \supseteq \text{dom}(h_r(r))$ if $s_r = s$ or $[\pi]^{\checkmark}_{\equiv'_\mathsf{n}}$.

If we have $[\pi]^{\lightning}_{\equiv'_{\mathsf{n}}}$, we also have $[\pi]^{\lightning}_{\equiv_{\mathsf{n}}}$. Thus, according to Definition 3.21 we then only need to regard $r \in [\pi]_{\equiv_{\mathsf{n}}}$ with $s_r = s$. Thus, we have $\tilde{h}(\tilde{s}|_{\pi}) = \widetilde{f} \in$ Instances with $\mathrm{dom}(\widetilde{f}') \supseteq \mathrm{dom}(\widetilde{f})$.

(j) We have $\widetilde{h}'(\tilde{s}'|_{\pi}) = (\tilde{i}'_l, \widetilde{f}') \in$ Arrays.

According to Definition 3.21, for each r as above we have $\overleftarrow{h}_r(\overleftarrow{s}_r|_{\pi'}) = f' \in$ Instances with $\mathrm{dom}(f') = \emptyset$ or $\overleftarrow{h}_r(\overleftarrow{s}_r|_{\pi'}) \in$ Arrays. Let f' with $\overleftarrow{h}_r(\overleftarrow{s}_r|_{\pi'}) = (i'_l, f') \in$ Arrays or $\overleftarrow{h}_r(\overleftarrow{s}_r|_{\pi'}) = f' \in$ Instances, and let f with $h_r(r) = (i_l, f) \in$ Arrays or $h_r(r) = f' \in$ Instances.

We have $\mathrm{dom}(\widetilde{f}') \supseteq \mathrm{dom}(f')$ if $\overleftarrow{s}_r = s'$ or $[\pi]^{\checkmark}_{\equiv'_{\mathsf{n}}}$. With Definition 3.25(i,j) we also have $\mathrm{dom}(\widetilde{f}') \supseteq \mathrm{dom}(f)$ if $s_r = s$ or $[\pi]^{\checkmark}_{\equiv'_{\mathsf{n}}}$.

If we have $[\pi]^{\lightning}_{\equiv'_{\mathsf{n}}}$, we also have $[\pi]^{\lightning}_{\equiv_{\mathsf{n}}}$. Thus, according to Definition 3.21 we then only need to regard $r \in [\pi]_{\equiv_{\mathsf{n}}}$ with $s_r = s$. Thus, we have $\tilde{h}(\tilde{s}|_{\pi}) = \widetilde{f} \in$ Instances with $\mathrm{dom}(\widetilde{f}) = \emptyset$, or $\tilde{h}(\tilde{s}|_{\pi}) = (\tilde{i}_l, \tilde{f}) \in$ Arrays with $\mathrm{dom}(\tilde{f}) \subseteq \mathrm{dom}(\tilde{f}')$.

Let $\{\pi, \pi'\} \in \mathrm{SPos}(\tilde{s}')$.

(k) We have $\tilde{s}'|_{\pi} \neq \tilde{s}'|_{\pi'}$ and $\pi, \pi' \in \mathrm{SPos}(\tilde{s})$. With $[\pi]_{\equiv'_{\mathsf{n}}} \neq [\pi']_{\equiv'_{\mathsf{n}}}$ and Lemma 3.11 we also have $[\pi]_{\equiv_{\mathsf{n}}} \neq [\pi']_{\equiv_{\mathsf{n}}}$. Thus, the claim follows.

(l) We have $\tilde{s}'|_{\pi} = \tilde{s}'|_{\pi'}$, $\tilde{h}'(\tilde{s}'|_{\pi}) \in$ Instances $\cup$ Arrays, and $\pi, \pi' \in \mathrm{SPos}(\tilde{s})$. If $[\pi]_{\equiv_{\mathsf{n}}} = [\pi']_{\equiv_{\mathsf{n}}}$, the claim follows. Thus, we only consider the case that $[\pi]_{\equiv_{\mathsf{n}}} \neq [\pi']_{\equiv_{\mathsf{n}}}$ and show that $\tilde{s}|_{\pi} =^{?} \tilde{s}|_{\pi'}$.

As $\tilde{s}'|_{\pi} = \tilde{s}'|_{\pi'}$, we also have $[\pi]_{\equiv'_{\mathsf{n}}} = [\pi']_{\equiv'_{\mathsf{n}}}$. Let $s_r|_{\pi_1} = r \in [\pi]_{\equiv_{\mathsf{n}}}$ and $s_{r'}|_{\pi_2} = r' \in [\pi']_{\equiv_{\mathsf{n}}}$. With Lemma 3.14 we have $\overleftarrow{s}_r|_{\pi_1} \in [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s}_{r'}|_{\pi_2} \in [\pi']_{\equiv'_{\mathsf{n}}}$, thus $\overleftarrow{s}_r|_{\pi_1} \equiv'_{\mathsf{n}} \overleftarrow{s}_{r'}|_{\pi_2}$. If we have $s_r = s_{r'}$, we also have $\overleftarrow{s}_r = \overleftarrow{s}_{r'}$. Thus, with Definition 3.17 we have $\overleftarrow{s}_r|_{\pi_1} = \overleftarrow{s}_{r'}|_{\pi_2}$ or $\overleftarrow{s}_r|_{\pi_1} =^{?} \overleftarrow{s}_{r'}|_{\pi_2}$. With Definition 3.25(l,q) we have $r =^{?} r'$.

Assume we have $s_{r_a}|_{\pi_a} = r_a \xrightarrow{\tau_a} [\pi]_{\equiv_{\mathsf{n}}}$ and $s_{r_b}|_{\pi_b} = r_b \xrightarrow{\tau_b} [\pi']_{\equiv_{\mathsf{n}}}$. As $[\pi]_{\equiv_{\mathsf{n}}} \neq [\pi']_{\equiv_{\mathsf{n}}}$, we know $r_a \neq r_b$ or $\tau_a \neq \tau_b$. With Lemmas 3.10 and 3.14 we then also have $\overleftarrow{s}_{r_a}|_{\overline{\pi_a\tau_a}} \mathrel{-\!\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s}_{r_b}|_{\overline{\pi_b\tau_b}} \mathrel{-\!\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$. With Definition 3.25(m,r) and Definition 3.17(iii,v) we then also have $r_a \mathrel{\bigvee\!\!\!\!/} r_b$ if $s_{r_a} = s_{r_b}$.

Now consider that we have $s_{r'}|_{\pi_a} \xrightarrow{\tau_a} [\pi]_{\equiv_{\mathsf{n}}}$, thus we also have $\overleftarrow{s}_{r'}|_{\overline{\pi_a\tau_a}} \mathrel{-\!\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$. If $\overleftarrow{s}_{r'}|_{\pi_a\tau_a} \in [\pi]_{\equiv'_{\mathsf{n}}}$, with Definition 3.17(ii) and Definition 3.25(m) we have $s_{r'}|_{\pi_a} \mathrel{\bigvee\!\!\!\!/} r'$. If $\overleftarrow{s}_{r'}|_{\overline{\pi_a\tau_a}} \rightarrow [\pi]_{\equiv'_{\mathsf{n}}}$, with Definition 3.17(iii) we have $\overleftarrow{s}_{r'}|_{\overline{\pi_a\tau_a}} \mathrel{\bigvee\!\!\!\!/} r'$. Thus, with Definition 3.25(r) we also have $s_{r_a}|_{\pi_a} \mathrel{\bigvee\!\!\!\!/} r'$. Similarly, for $s_r|_{\pi_b} \xrightarrow{\tau_b} [\pi']_{\equiv_{\mathsf{n}}}$ we have $s_r|_{\overline{\pi_b\tau_b}} \mathrel{\bigvee\!\!\!\!/} r$.

Thus, if we have $\overleftarrow{s}_r = \overleftarrow{s}_{r'}$, with Definition 3.23(b) we have $\tilde{s}|_{\pi} =^{?} \tilde{s}|_{\pi'}$. Now we consider the case that for all $s_r|_{\pi_1} = r \in [\pi]_{\equiv_{\mathsf{n}}}$ and $s_{r'}|_{\pi_2} = r' \in [\pi']_{\equiv_{\mathsf{n}}}$ we have

$s_r \neq s_{r'}$. According to Lemma 3.16 this case is not possible. Thus, the claim is shown.

(m) We have $\tilde{s}'|_\pi = \tilde{s}'|_{\pi'}$ or $\tilde{s}'|_\pi =^? \tilde{s}'|_{\pi'}$ with $\pi \neq \pi'$. We also have $\tilde{h}'(\tilde{s}'|_\pi) \in$ $\textsc{Instances} \cup \textsc{Arrays}$ and $\{\pi, \pi'\} \not\subseteq \mathrm{SP}\textsc{os}(\tilde{s})$. Furthermore, we may assume $\tilde{s}|_{\overline{\pi}} \neq \tilde{s}|_{\overline{\pi'}}$ or π, π' have different suffixes w.r.t. $\tilde{s}$. Thus, we need to show $\tilde{s}|_{\overline{\pi}} \mathbin{\backslash\!\!/} \tilde{s}|_{\overline{\pi'}}$. W.l.o.g. assume $\pi \notin \mathrm{SP}\textsc{os}(\tilde{s})$. Then the claim directly follows from Lemma 3.23.

(n) We have $\{\alpha\tau, \alpha\tau'\} \subseteq \mathrm{SP}\textsc{os}(\tilde{s}')$ with $\tilde{h}'(\tilde{s}'|_{\alpha\tau}) \in \textsc{Instances} \cup \textsc{Arrays}$, $\tilde{s}'|_{\alpha\tau} = \tilde{s}'|_{\alpha\tau'}$, and $\tau \neq \varepsilon$. Furthermore τ, τ' have no common intermediate reference from α in $\tilde{s}'$.

With $\tilde{s}'|_{\alpha\tau} = \tilde{s}'|_{\alpha\tau'}$ we have $[\alpha\tau]_{\equiv'_{\mathsf{n}}} = [\alpha\tau']_{\equiv'_{\mathsf{n}}}$. Let $s_{r_\alpha}|_{\mathring{\alpha}} = r_\alpha \mathrel{-\!\!\in} [\overline{\alpha}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$ with $\alpha = \overline{\alpha}_{\tilde{s}}\beta$. With Lemma 3.19 we have $\overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta\tau}} \mathrel{-\!\!\in} [\alpha\tau]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta\tau'}} \mathrel{-\!\!\in} [\alpha\tau']_{\equiv'_{\mathsf{n}}}$. Thus, we also have $\overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta}} \overset{\mathring{\beta}\tau}{\twoheadrightarrow} [\alpha\tau]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta}} \overset{\mathring{\beta}\tau'}{\twoheadrightarrow} [\alpha\tau']_{\equiv'_{\mathsf{n}}}$ where $\mathring{\alpha}\beta = \overline{\mathring{\alpha}\beta}_{\overleftarrow{s_{r_\alpha}}}\mathring{\beta}$. With Definition 3.17(iv) we have $\overleftarrow{s_{r_\alpha}}|_{\mathring{\alpha}\beta\tau} = \overleftarrow{s_{r\alpha}}|_{\mathring{\alpha}\beta\tau'}$, or $\overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta}} \mathbin{\backslash\!\!/} \overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta}}$ and, if $\tau' = \varepsilon$, also $\overleftarrow{s_{r_\alpha}}|_{\overline{\mathring{\alpha}\beta}} \circlearrowleft_{F'_i}$ with $F'_i \subseteq \tau$. Then, with Definition 3.25(n,o,r) we have $s_{r_\alpha}|_{\mathring{\alpha}} \mathbin{\backslash\!\!/} s_{r_\alpha}|_{\mathring{\alpha}}$ and, if $\tau' = \varepsilon$, also $s_{r_\alpha}|_{\mathring{\alpha}} \circlearrowleft_{F_i}$ with $F_i \subseteq \tau$.

(o) We have $\tilde{s}'|_\pi \circlearrowleft_{F'}$. With Definition 3.23(g,h) we have $F' = \bigcup_i F'_i$ with F'_i as follows. We have $\overleftarrow{s_{\mathring{r}}}|_{\pi'} \circlearrowleft_{F'_i}$ for all $\overleftarrow{s_{\mathring{r}}}|_{\pi'} = \mathring{r} \mathrel{-\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$. If $[\pi]^{\lightning}_{\equiv'_{\mathsf{n}}}$, we only regard the cases where $\overleftarrow{s_{\mathring{r}}} = s'$.

Let τ with $\overline{\pi}_{\tilde{s}}\tau = \pi$ and consider any $s_r|_{\pi_1} = r \overset{\tau_1}{\mathrel{-\!\!\in}} [\overline{\pi}_{\tilde{s}}]_{\equiv_{\mathsf{n}}}$ where $s_r = s$ if $[\overline{\pi}_{\tilde{s}}]^{\lightning}_{\equiv_{\mathsf{n}}}$. With Lemma 3.19 we have $\overleftarrow{s_r}|_{\overline{\pi_1\tau_1}} \mathrel{-\!\!\in} [\overline{\pi}_{\tilde{s}}]_{\equiv'_{\mathsf{n}}}$ and $s_{\overleftarrow{\overline{\pi_1\tau_1\tau}}} \mathrel{-\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$. Thus, we also have $s_{\overleftarrow{\overline{\pi_1\tau_1\tau}}} \circlearrowleft_{F'_i}$ with $F'_i \subseteq F'$ if $s_r = s$ or $[\overline{\pi}_{\tilde{s}}]^{\checkmark}_{\equiv_{\mathsf{n}}}$. With Definition 3.23(g,h) and Definition 3.25(o) we then have $s_r|_{\pi_1} \circlearrowleft_{F_i}$ with $F_i \subseteq F'$. Combined, we have $\tilde{s}|_{\overline{\pi}} \circlearrowleft_F$ with $F \subseteq F'$.

(p) We have $\tilde{s}'|_\pi?$ and $\pi \in \mathrm{SP}\textsc{os}(\tilde{s})$. According to Lemma 3.14 for each $r \in [\pi]_{\equiv_{\mathsf{n}}}$ with $s_r|_{\pi'} = r$ we have $\overleftarrow{s_r}|_{\pi'} \in [\pi]_{\equiv'_{\mathsf{n}}}$. With Definition 3.23(a) we have $\overleftarrow{s_r}|_{\pi'}?$ for all such r. Thus, with Definition 3.25(p) we also have $s_r|_{\pi'}?$. Then the claim follows with Definition 3.23(a).

(q) We have $\tilde{s}'|_\pi =^? \tilde{s}'|_{\pi'}$, thus $\tilde{s}'|_\pi \neq \tilde{s}'|_{\pi'}$. We also have $\{\pi, \pi'\} \subseteq \mathrm{SP}\textsc{os}(\tilde{s})$. Consider any $s_r|_{\pi_1} = r \in [\pi]_{\equiv_{\mathsf{n}}}$ and $s_{r'}|_{\pi_2} = r' \in [\pi']_{\equiv_{\mathsf{n}}}$ with $s_r = s_{r'}$. With Lemma 3.14 we have $\overleftarrow{s_r}|_{\pi_1} \in [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s_r}|_{\pi_2} \in [\pi']_{\equiv'_{\mathsf{n}}}$. According to Definition 3.23(b) we have $\overleftarrow{s_r}|_{\pi_1} =^? \overleftarrow{s_r}|_{\pi_2}$. With Definition 3.25(q) we then also have $s_r|_{\pi_1} =^? s_r|_{\pi_2}$.

Assume we have $s_{r_a}|_{\pi_a} = r_a \overset{\tau_a}{\twoheadrightarrow} [\pi]_{\equiv_{\mathsf{n}}}$ and $s_{r_b}|_{\pi_b} = r_b \overset{\tau_b}{\twoheadrightarrow} [\pi']_{\equiv_{\mathsf{n}}}$. As $[\pi]_{\equiv_{\mathsf{n}}} \neq [\pi']_{\equiv_{\mathsf{n}}}$, we know $r_a \neq r_b$ or $\tau_a \neq \tau_b$. With Lemmas 3.10 and 3.14 we then also have $\overleftarrow{s_{r_a}}|_{\overline{\pi_a\tau_a}} \mathrel{-\!\!\in} [\pi]_{\equiv'_{\mathsf{n}}}$ and $\overleftarrow{s_{r_b}}|_{\overline{\pi_b\tau_b}} \mathrel{-\!\!\in} [\pi']_{\equiv'_{\mathsf{n}}}$. With Definition 3.23(b), Definition 3.25(m,r), and Definition 3.17(iii,v) we then also have $r_a \mathbin{\backslash\!\!/} r_b$ if $s_{r_a} = s_{r_b}$.

Now consider that we have $s_{r'}|_{\pi_a} \xrightarrow{\tau_a} [\pi]_{\equiv_n}$. Then we also have $\overleftarrow{s}_{r'}|_{\overline{\pi_a\tau_a}} \mathrel{-\!\!\in} [\pi]_{\equiv'_n}$. If $\overleftarrow{s}_{r'}|_{\pi_a\tau_a} \in [\pi]_{\equiv'_n}$, with Definition 3.23(b) and Definition 3.25(m) we have $s_{r'}|_{\pi_a} \bigvee r'$. If $\overleftarrow{s}_{r'}|_{\overline{\pi_a\tau_a}} \rightarrow [\pi]_{\equiv'_n}$, with Definition 3.23(b) we have $\overleftarrow{s}_{r'}|_{\overline{\pi_a\tau_a}} \bigvee r'$. Thus, with Definition 3.25(r) we also have $s_{r'}|_{\pi_a} \bigvee r'$. Similarly, for $s_r|_{\pi_b} \xrightarrow{\tau_b} [\pi']_{\equiv_n}$ we have $s_r|_{\overline{\pi_b\tau_b}} \bigvee r$.

Thus, with Definition 3.23(b) we have $\tilde{s}|_\pi =^? \tilde{s}|_{\pi'}$.

(r) We have $\tilde{s}'|_\pi \bigvee \tilde{s}'|_{\pi'}$.

We first consider the case that $\pi = \pi'$. Let $s_r|_{\pi_1} = r \xrightarrow{\tau_1}\!\in [\overline{\pi}_{\tilde{s}}]_{\equiv_n}$ with $s_r = s$ if $[\overline{\pi}_{\tilde{s}}]^{\not\checkmark}_{\equiv_n}$. With Lemma 3.19 we have $\overleftarrow{s}_r|_{\overline{\pi_1\tau_1}} \mathrel{-\!\!\in} [\overline{\pi}_{\tilde{s}}]_{\equiv'_n}$. We also have $\overleftarrow{s}_r|_{\overline{\pi_1\tau_1\tau}} \mathrel{-\!\!\in} [\pi]_{\equiv'_n}$ for $\pi = \overline{\pi}_{\tilde{s}}\tau$. According to Definition 3.23(e,f) we have $\overleftarrow{s}_r|_{\overline{\pi_1\tau_1\tau}} \bigvee \overleftarrow{s}_r|_{\overline{\pi_1\tau_1\tau}}$ if $s_r = s$ or $[\overline{\pi}_{\tilde{s}}]^{\checkmark}_{\equiv_n}$. Thus, we also have $s_r|_{\pi_1} \bigvee s_r|_{\pi_1}$ and the claim follows with Definition 3.23(e,f).

Now consider that $\pi \neq \pi'$. Let $s_r|_{\pi_1} = r \in [\overline{\pi}_{\tilde{s}}]_{\equiv_n}$ and $s_{r'}|_{\pi_2} = r' \in [\overline{\pi'}_{\tilde{s}}]_{\equiv_n}$ with $s_r = s_{r'}$. With Lemma 3.19 we have $\overleftarrow{s}_r|_{\overline{\pi_1\tau}} \mathrel{-\!\!\in} [\pi]_{\equiv'_n}$ and $\overleftarrow{s}_r|_{\overline{\pi_2\tau'}} \mathrel{-\!\!\in} [\pi']_{\equiv'_n}$ where $\pi = \overline{\pi}_{\tilde{s}}\tau$ and $\pi' = \overline{\pi'}_{\tilde{s}}\tau'$. With Definition 3.23(e,f) we have $\overleftarrow{s}_r|_{\overline{\pi_1\tau}} \bigvee \overleftarrow{s}_r|_{\overline{\pi_2\tau'}}$ if $s_r = s$, or $[\overline{\pi}_{\tilde{s}}]^{\checkmark}_{\equiv_n}$ and $[\overline{\pi'}_{\tilde{s}}]^{\checkmark}_{\equiv_n}$. Thus, we also have $s_r|_{\pi_1} \bigvee s_r|_{\pi_2}$ and the claim follows with Definition 3.23(e,f).

(s) This directly follows from Definition 3.23 and Definition 3.25(s).

Thus, the claim holds. □

3.5. Symbolic Execution Graphs for Recursive Programs

We now show that the edges used in the graph are sound, similar to Theorem 1.56. In the case of *instance edges* this is shown by proving that the $\sqsubseteq$ relation is transitive.

Using Theorem 3.7, transitivity of $\sqsubseteq$ can now easily be proved by reducing it to the case of states with call stacks of the same size. For this case, we proved transitivity of $\sqsubseteq$ already in Theorem 1.4.

Lemma 3.24 ($\sqsubseteq$ is transitive) Let s, s', s'' be states with $s_1 \sqsubseteq s_2$ and $s_2 \sqsubseteq s_3$. Then $s_1 \sqsubseteq s_3$.

Proof. We have $|s_1| \geq |s_2| \geq |s_3|$. From $s_1 \sqsubseteq s_2$ we can conclude that there is a state $\tilde{s}'$ that can be obtained by repeated context concretization of s_2 such that $|s_1| = |s_2|$ and $s_1 \sqsubseteq \tilde{s}'$. Let $\tilde{s}$ be the state resulting from s_3 by performing the same context concretizations. Thus, $|\tilde{s}'| \geq |\tilde{s}|$ and, by Theorem 3.7, we have $\tilde{s}' \sqsubseteq \tilde{s}$. Hence, by further context concretization of $\tilde{s}$, we can obtain a state $\tilde{\tilde{s}}$ with $|\tilde{s}'| = |\tilde{\tilde{s}}|$ and $\tilde{s}' \sqsubseteq \tilde{\tilde{s}}$. Hence, we now have $|s_1| = |\tilde{s}'| = |\tilde{\tilde{s}}|$ and $s_1 \sqsubseteq \tilde{s}' \sqsubseteq \tilde{\tilde{s}}$. Thus, Theorem 1.4 implies $s_1 \sqsubseteq \tilde{\tilde{s}}$. Since $\tilde{\tilde{s}}$ was obtained by repeated context concretization from s_3, this also implies $s_1 \sqsubseteq s_3$. □

After having shown the soundness of instance edges, we now prove the soundness of *context concretization edges*.

Lemma 3.25 (Soundness of Context Concretization Edges) Let $c \sqsubseteq s$ for a return state s and $c \xrightarrow{jvm} c'$. Then there exists a context concretization $\tilde{s}$ of s with a call state $\ddot{s}$ such that $c \sqsubseteq \tilde{s}$ and where s and $\tilde{s}$ are connected with a context concretization edge.

Proof. As $c \xrightarrow{jvm} c'$, c cannot be a program end. As the top stack frames of c and s are at the same program position, we obtain $|c| \geq 2$ and thus, $|c| > |s|$. Hence, according to Definition 3.25, there exists a state $\tilde{s}'$ obtained by repeated context concretization from s such that $|c| = |\tilde{s}'|$ and $c \sqsubseteq \tilde{s}'$. Since $|c| > |s|$, we must perform at least one context concretization step from s to $\tilde{s}'$. Let $\tilde{s}$ be the result of performing the first of these context concretizations on s. Then, by Definition 3.25, we also have $c \sqsubseteq \tilde{s}$. □

To prove soundness of *input arguments creation edges*, we need to consider the fact that input arguments are added to the topmost stack frame. However, according to Definition 3.25(s) we may only have $c \sqsubseteq \ddot{s}$ for a concrete state c and $|c| = |\ddot{s}|$ if there is no input argument in $\ddot{s}$. However, we know that from every call state $\ddot{s}$ we have an outgoing *call edge* leading to a state s' where the lower stack frames are not represented.

Assume we have $c \xrightarrow{jvm} c'$ with $c \sqsubseteq s$. If s is an invoking state and $\ddot{s}$ is the corresponding call state, thus s is connected to $\ddot{s}$ using an input arguments creation edge, we also consider the state s' where $\ddot{s}$ is connected to s' using a call edge. Then we show $c' \sqsubseteq s'$. As $|c'| > |s'| = 1$, we must use context concretization to show $c' \sqsubseteq s'$, thus we do not need input arguments in c'.

Lemma 3.26 (Soundness of Input Arguments Creation and Call Edges) Let s be an invoking state, let $\ddot{s} = (\langle \ddot{fr}_0, \ldots, \ddot{fr}_m \rangle, \ddot{h}, \ddot{t}, \ddot{hp}, \ddot{sf}, \bot, \ddot{ic}, \bot)$ be the corresponding call state, and let $s' = (\langle \ddot{fr}_0 \rangle, \ddot{h}, \ddot{t}, \ddot{hp}, \ddot{sf}, \bot, \ddot{ic}, \bot)$ be the corresponding call abstraction state. Then $s \sqsubseteq s'$.

Proof. Context concretization of s' with $\ddot{s}$ results in a state that is identical to $\ddot{s}$ (up to renaming of variables). This implies $\ddot{s} \sqsubseteq s'$ by Definition 3.25. As s and $\ddot{s}$ only differ by the input arguments added to the top stack frame, we also have $s \sqsubseteq s'$. □

It remains to show that also *refinement edges* and *evaluation* edges are correct. However, as the proof of Theorem 3.7 already is very complex, we decided to not also formally show correctness for these edges. Furthermore, in [BOG11] (Lemmas 11, 12, and 14) we already have shown correctness of these edges in a similar setting. Thus, the interested reader may adapt these proofs to the setting of this thesis.

In the case of evaluation edges, similar to the situation in Chapter 1, the PUTFIELD opcode is of special interest. As evaluation of PUTFIELD may alter information also visible in lower stack frames, which is not explicitly represented in this analysis, we must take care to regard these changes using other means. In this setting, the created input arguments represent the references of the lower stack frames. Thus, whenever a reference r is modified using PUTFIELD and we have an input argument $(\lambda, \gamma, \checkmark/\lightning)$ with $\lambda \rightsquigarrow r$, we mark the input argument as changed (thus, in the successor state we have $(\lambda', \gamma', \lightning)$ with $\gamma = \gamma'$). As the information of changed input arguments is regarded in context concretization, soundness follows.

Conjecture 3.27 (Soundness of Refinement Edges) Let c be a concrete state with $c \sqsubseteq s$ and let $\text{refine}(s) = \{s_1, \ldots, s_n\}$. Then there exists a state $s_i \in \text{refine}(s)$ with $c \sqsubseteq s_i$.

Conjecture 3.28 (Soundness of Evaluation Edges) Let c be a concrete state with $c \xrightarrow{jvm} c'$ and $c \sqsubseteq s$. If $s \xrightarrow{\text{EVAL}} s'$ then we have $c' \sqsubseteq s'$.

3.5.1 Graph Construction

Using the concepts introduced in this chapter, most notably context concretization, we now show how Symbolic Execution Graphs are constructed for recursive methods. In

Algorithm 16 we show the updated version of EVALUATE as used in Algorithm 1. Here, we only need to consider three changes:

- When evaluating opcodes like PUTFIELD, we need to also mark input arguments as changed (cf. line 20). Otherwise, we can evaluate just as in Chapter 1.
- When a new method is invoked, i.e., we have an invoking state, we create the corresponding call state by adding input arguments to the state (cf. lines 1–4). Afterwards, for the call state we create a state consisting only out of the topmost stack frame (cf. lines 5-8)
- If we encounter a return state, we perform context concretization with all call states and add the resulting states to the graph. However, as this may result in infinitely many return states with infinitely many states resulting out of context concretization, using FORCEABSTRACTION we take care that this cannot happen (cf. lines 9–17).

Algorithm 16: Evaluation

Input: $s \in$ STATES, Symbolic Execution Graph $\mathcal{G}$
1: **if** s is an invoking state **then**
2:    create state s' as a copy of s
3:    add input arguments to s' according to Definition 3.6
4:    connect s to s' using an *input arguments creation edge*
5: **else if** s is a call state **then**
6:    create state s' as a copy of s
7:    remove all but the topmost stack frame of s'
8:    connect s to s' using a *call edge*
9: **else if** s is a return state **then**
10:    **if** there is a return state s' of the same shape **then**
11:       **if** $s \sqsubseteq s'$ **then**
12:          connect s to s' using an instance edge
13:       **else**
14:          FORCEABSTRACTION(s, s')
15:    **else**
16:       **for all** call states $\ddot{s}$ with $\mathrm{cc}(s, \ddot{s}) = \tilde{s}$ **do**
17:          connect s to $\tilde{s}$ using a context concretization edge
18: **else**
19:    EVALUATE as in Chapter 1
20:    if necessary, mark input arguments as changed

As even in the case of recursive methods no state may contain an unbounded number of stack frames, Algorithm 16 is better suited for recursive programs when compared to the analysis in Chapter 1. However, we still need to ensure that only a finite number of input arguments exists in each state. This is addressed in Section 3.6.

We now finally prove the soundness of Symbolic Execution Graphs, i.e., that every concrete JAVA BYTECODE evaluation corresponds to a computation path in the Symbolic Execution Graph.

Theorem 3.29 (Soundness of Symbolic Execution Graphs) Let c, c' be concrete states with $c \xrightarrow{jvm} c'$. If a Symbolic Execution Graph contains a state s with $c \sqsubseteq s$, then the graph contains a path from s to a state s' with $c' \sqsubseteq s'$.

Proof. We prove the theorem by induction on the sum of the lengths of all paths from s to the next evaluation edge. This sum is always finite, since we required that every cycle of a Symbolic Execution Graph must contain at least one evaluation edge. We perform a case distinction on the type of the outgoing edges of s.

If s has an *instance edge* to s', then $s \sqsubseteq s'$ and by Lemma 3.24 we also have $c \sqsubseteq s'$ and the claim follows from the induction hypothesis. If s has an *input arguments creation edge* to $\ddot{s}$, we also have a *call edge* from $\ddot{s}$ to s'. Thus, with Lemma 3.26 we have $c \sqsubseteq s'$ and the claim follows from the induction hypothesis. If the outgoing edges of s are *context concretization* or *refinement edges* to $s_1, \ldots, s_n$, we know by Lemma 3.25 and Conjecture 3.27 that there is an s_i with $c \sqsubseteq s_i$. Again, then the claim follows from the induction hypothesis. Finally, if there is an *evaluation* edge from s to s', we know by Conjecture 3.28 that $c' \sqsubseteq s'$. ☐

3.6. Abstraction of Input Arguments

According to Definition 3.6 we need to add input arguments to the state whenever a recursive method is invoked. If the number of input arguments is bounded, the graph construction as shown in Section 3.5.1 terminates. However, there may be programs for which an unbounded number of input arguments is necessary. Indeed, in Example 3.30 a simple algorithm is shown for which the graph construction as presented in this chapter does not terminate.

Example 3.30 The algorithm `length` recursively computes the length of the current `List`. For the invocation in line 8, input arguments are created both for `next` (which is an argument of the invoked method) and `this` (which is a predecessor of `next`). Furthermore, which is crucial, we also need to add an input argument for any input argument we already created for previous calls, as the corresponding reference also is a predecessor of `next`.

```
public class List {
  List next;

  int length() {
    if (next == null) {
      return 1;
    }
    return 1 + next.length();
  }
}
```

One could optimize Definition 3.6 such that less input arguments need to be created. However, there always are cases where one needs to create additional input arguments similar to the situation shown in Example 3.30.

In order to have a finite graph construction, one possibility is to *merge* two or more input arguments into one input argument. In Example 3.30 it might suffice to only create an input argument representing *any* predecessor of the current list element, so that a single input argument suffices to represent the information of an unbounded number of stack frames.

However, in order for this idea to work, several changes to the technique presented so far would be necessary:

Instance Definition Assume that from s to s' we merge two input arguments into one with the intention that $s \sqsubseteq s'$ holds. Then in most cases in s' the merged input argument is represented by a single reference r, while in s the two input arguments are represented by two different references $r' \neq r''$. Thus, Definition 3.25(k) would need to be adapted accordingly. One idea is that Definition 3.25(k) does not need to hold for the case of merged input arguments.

Evaluation We must take care that any reference representing a merged input argument cannot be used in the actual computation. For example code corresponding to $\mathsf{if}\ (\mathsf{x} == \mathsf{x})$ is problematic, if the reference used for x may represent two *different* object instances on the heap. Even if the reference used for a merged input argument is only reachable using heap predicates, using refinement it may be possible to create a situation where this reference is made available on the operand stack. Thus, we may need to have special rules for example how $=^?$ is treated in the case of merged input arguments.

Write Accesses When, for example, evaluating PUTFIELD, it may be necessary to treat merged input arguments differently.

Context Concretization When returning from a method containing a merged input argument, we need to *split* (un-merge) the merged input argument so that the reference corresponding to the stack frame we return to is represented, but we also still have the (merged) input argument corresponding to the (unbounded) number of lower stack frames.

In [RC11] the authors discuss this problem in another setting, where *cutpoints* correspond to our definition of input arguments.

To summarize, input arguments and their usage as demonstrated in this chapter seem to be a great help in constructing Symbolic Execution Graphs for recursive programs. However, the technique still needs to be adapted so that only a finite number of input arguments is necessary. The idea of merging input arguments may be helpful with this.

3.7. Conclusion and Outlook

Using the technique presented in Chapter 1, for recursive programs the construction might not terminate as states with an unbounded number of stack frames could be created. In this chapter, we presented an extension that enables call stack abstraction. With this, it is possible to also represent recursive programs using states with a bounded number of stack frames. In order to reason about the necessary information lost in the abstraction, we introduced *input arguments* and the concept of *context concretization*.

As the main contribution of this chapter, we have shown that using context concretization it indeed is possible to introduce call stack abstraction and still create Symbolic Execution Graphs with the desired correctness properties. Here the main challenge was the extension of the formalization already shown in [BOG11] so that context concretization as presented in this thesis also works on states making use of heap predicates.

However, the presented technique does not suffice to guarantee construction of a *finite* Symbolic Execution Graph for recursive programs (as discussed in Section 3.6). As such, corresponding adoptions are left for future work.

The Symbolic Execution Graphs created using the techniques presented in this thesis are created in the context of the whole program, meaning that every opcode (and class and method) that may be executed is known in advance. While the states created for different methods form clusters in the graph (which only are connected using *call edges* and *context concretization edges*), the creation of input arguments still depends on the individual call states. Thus, it is not possible to analyze a method without having detailed information about all possible call sites. As a consequence, in order to analyze code making use of, for example, collection classes in java.util, we also need to analyze these library classes.

Analysis of library classes which are known prior to the analysis and which do not change seems to be redundant. Furthermore, preliminary experiments have shown that the analysis does not scale to larger programs making use of many different methods (and classes).

For future work, one idea is to make the construction of Symbolic Execution Graphs more modular by abstracting the information currently represented in call states and carried over to the invoked method (in the form of input arguments). A discussion of this idea and first results are presented in [Fro13]. Furthermore, in this thesis we left out a possible optimization regarding static fields. According to Definition 3.23 we disregard information about static fields in the call state $\ddot{s}$. Instead, one could simply extend Definition 3.23 such that also information about static fields in the call state is retained, if that static field is known to be left unchanged by the invoked method.

4. Bug Detection

The Symbolic Execution Graphs as presented in Chapters 1 and 3 can not only be used to prove termination, but also contain a lot of detailed information that can easily be used for other analyses. In this chapter we present a technique that helps to find bugs which are hard to find without having as detailed information.

This technique is based on the idea of running a precise dead code analysis on the Symbolic Execution Graph constructed for a program. Then, instead of eliminating dead code, we show parts of the program containing dead code to the user. We assume that most (if not all) code is indeed intended to have a purpose. Because of this, bugs leading to dead code can be identified using this method.

Example 4.1 (Bug) Assume that the method createGraph() creates new graph objects so that graphOne and graphTwo do not share on the heap. Then the algorithm adds all nodes from graphOne which are connected to a node stored in the variable source to graphTwo. In Example 4.18 on page 200 you can also find the complete code of this example.

```
1 Graph graphOne = createGraph();
2 Graph graphTwo = createGraph();
3
4 Node source = graphTwo.getRootNode();
5 for (Node node : graphOne.getNodes()) {
6   if (areConnected(source, node)) {
7     graphTwo.addNode(node);
8   }
9 }
```

Due to a bug the source node is taken from graphTwo, so that the following calls to areConnected for node from graphOne and source from graphTwo always return false. Because of this the intended operation of adding node to graphTwo never is executed.

If the user is interested in the value of graphTwo, our analysis is able to give the information that the code in lines 4–9 can be ignored for this purpose. A user, confronted with this result, could easily see that it *should* influence the intended result and then find and fix the bug mentioned above.

To find bugs as shown in Example 4.1, the analysis must be able to have precise information about the heap. In the example, it must be known that source and node do not share. While gathering this information makes other approaches more complicated, we can just use the information already available in Symbolic Execution Graphs. Furthermore, this technique also benefits from most optimizations in the construction of Symbolic Execution Graphs. For example, if stronger heap predicates are introduced, the Symbolic Execution Graph contains more detailed information and this technique automatically provides better results.

Goal of the analysis

The goal of most dead code analyses is to identify code that can be removed without influencing the result of the computation. A more formal definition is given in Conjecture 4.17, which is introduced after explaining the details of the analysis.

In classical dead code analyses, for example those implemented in optimizing compilers that remove dead code, code influencing the termination behavior must not be considered dead code [Ben05].

In the technique presented here, we do not remove code and, therefore, may ignore the fact that code can influence the termination behavior of a program. Furthermore, as already introduced, there exist other techniques to prove or disprove termination which can be used if the user is interested in these questions.

Example 4.2 (Termination behavior) In this example, the loop including the call to someMethod may not terminate and, therefore, must not be considered as dead code intended for removal. However, if we ignore possible non-termination and are only interested in code influencing the value of res, we may conclude that lines 2–5 are not relevant.

```
1 int res = 2;
2 int i = 0;
3 while (someMethod(i)) {
4   i++;
5 }
6 return res;
```

The goal of this analysis is to identify dead code that can be considered irrelevant for desired outcomes. For that, the user needs to mark certain parts of the code as relevant. Based on this definition other parts of the program must consequently be marked as relevant and, finally, code not marked as relevant is considered irrelevant.

Example 4.3 (Relevance) In the following method, the sum of the arguments a and b is returned. Additionally the method contains a computation that is not needed to compute the sum.

```
1 public int add(int a, int b, int c) {
2   int res = a + b;
3   int temp = res/c;
4   return res;
5 }
```

If we define that only the return value of add is relevant, our analysis can provide the information that the computation of temp in line 3 is irrelevant.

However, we may also be interested in exceptions thrown by a method. In the example above, if we are interested in thrown exceptions, we must consider that the division res/c could throw an instance of ArithmeticException and, therefore, must not be considered irrelevant.

Structure

After discussing related work in Section 4.1, in Section 4.2 we present the core idea of this technique and explain which important aspects need to be regarded. Then, in Section 4.3, we extend the core idea to cover most aspects of Java and formalize the presented ideas, resulting in algorithms performing the analysis. In Sections 4.4 and 4.5 we discuss how to present the obtained results to the user and discuss optimization ideas. In Section 4.6 we present a conjecture which states how the results of this analysis can be interpreted. In Section 4.7 we demonstrate the power of the analysis. Finally, in Section 4.8 we conclude and discuss how the presented results can and should be extended.

4.1. Related Work

The technique presented here is a constraint-based, inter-procedural, and context-sensitive data flow analysis [NNH99]. Some concepts used in this chapter correspond to the idea of *program dependence graphs* as in [FOW87].

In the past decades many techniques reasoning about or transforming code have been developed. Most importantly, *dead code* analyses as part of compilers may compute similar results [ASU85].

In contrast to most analyses, this analysis concentrates on finding and reporting bugs to the programmer. In [WZKSL13] the authors present a technique which identifies and reports *unstable* code. Here, the programmer may have used constructs with undefined

```
int a = foo();
int b = a;
int res = -a;
b++;
return res;
```

(a) Java program

```
INVOKESTATIC foo()
ISTORE_1 // store to a
ILOAD_1 // load a
ISTORE_2 // store to b
ILOAD_1 // load a
INEG // -a
ISTORE_3 // store to res
IINC 2, 1 // b++
ILOAD_3 // load res
IRETURN // return res
```

(b) corresponding Java Bytecode

Figure 4.1.: Small Java example

behavior. As such, compilers with corresponding optimizations may remove such code, possibly not behaving as intended by the programmer.

Tools like FindBugs [AHM+08] and Coverty [BBC+10] also make use of static analysis to find bugs in programs. However, while these tools are focussed on providing results for large programs, the used analyses are less precise.

To our knowledge, the technique of this chapter is the first to heavily rely on a pre-computed analysis (viz. Symbolic Execution Graphs) in order to keep the analysis itself simple.

4.2. Basic Idea

We construct a simple propositional formula that can easily be used to provide the desired information. For that, we associate a propositional variable with each opcode in the program and add implications to the formula so that variables set to true in a minimal model of the formula identify relevant opcodes. To construct the formula, a simple fixed-point algorithm is run on the finished Symbolic Execution Graph.

As we create a formula which consists of conjunctions over implications, in this chapter we define that implications have a higher precedence than conjunctions. This eases readability of the presented formulas, i.e.

$$a \to b \wedge c \to d = (a \to b) \wedge (c \to d) \neq a \to (b \wedge c) \to d$$

In the following example we will introduce the idea informally. For that, we assume that for each opcode we know which data is the corresponding input and output. This information is not directly available in the Symbolic Execution Graph, so that in later sections we need to extend the analysis to also provide this information. Nevertheless, the example gives a first understanding of the fundamental ideas used in this chapter.

Example 4.4 (Formula construction) In Fig. 4.1a JAVA code is shown that negates a value returned by a method foo(). In Fig. 4.1b the corresponding JAVA BYTECODE with 10 opcodes is shown.

Assume that we have propositional variables Rel_x for each of those opcodes (i.e., $1 \leq x \leq 10$) used to encode that the corresponding opcode in line x is relevant. At the end of the analysis the values assigned to these variables are used to compute the result which is presented to the user.

Furthermore, assume for each such x the variables In_x and Out_x are used to denote the relevance of all inputs resp. outputs of the xth opcode.

Thus, in this example we use three variables (Rel_x, In_x, Out_x) for each opcode. Although we only are interested in the relevance of each opcode (for which we use Rel_x), in the case of opcodes with multiple inputs or outputs it is possible to only mark specific inputs or outputs as relevant. Details of this idea will be presented later.

Without the need for any further analysis, we can construct the following implications. These denote that, if the output of an opcode is relevant, also the opcode itself is relevant. Furthermore, if an opcode is relevant then also its inputs are relevant. As an example, for the ILOAD_1 instruction in line 5 we get $Out_5 \to Rel_5 \wedge Rel_5 \to In_5$.

$$
\begin{aligned}
& Rel_{10} \to In_{10} \wedge \\
\bigwedge_{2 \leq x \leq 9} & Out_x \to Rel_x \wedge Rel_x \to In_x \wedge \\
& Out_1 \to Rel_1
\end{aligned}
$$

In the main part of the analysis we connect these variables by analyzing which inputs correspond to which outputs. If an input is relevant, then also the output providing the value is relevant.

$$
\underbrace{In_2 \to Out_1 \wedge In_3 \to Out_2 \wedge In_4 \to Out_3}_{\text{call foo(), store result to a and b}} \wedge \underbrace{In_8 \to Out_4}_{\text{increment b}} \wedge
$$
$$
\underbrace{In_5 \to Out_2 \wedge In_6 \to Out_5 \wedge In_7 \to Out_6 \wedge In_9 \to Out_7 \wedge In_{10} \to Out_9}_{\text{compute -a, store to res, return res}}
$$

If the user decides to consider the result of the IRETURN opcode as relevant, this corresponds to setting Rel_{10} to $\top$ (true). Then, in all models of the formula also the variables corresponding to the relevance of the opcodes at positions $\{1, 2, 5, 6, 7, 9, 10\}$ must be set to $\top$. Because we only consider minimal models, the variables for the relevance of the other opcodes $\{3, 4, 8\}$ are set to $\bot$ (false), indicating that these are not relevant for the computation of res. Indeed, the opcodes $\{3, 4, 8\}$ correspond to the JAVA code int b = a; b++, which does not contribute to the returned value res.

As seen in the preceding example we need to identify the opcodes providing the inputs for succeeding opcodes. However, because we work on arbitrary JAVA BYTECODE instead of code compiled from JAVA source code, usage of intermediate values on the operand stack may be close to arbitrary.

Example 4.5 (Useless Input) Both the code in line 4 and the addition $(1 + 2)$ in line 3 produce the value 3 as output. However, the true origin for the returned value is the result of the IADD opcode, as the value introduced in line 4 is removed in line 5.

```
1  ICONST_1
2  ICONST_2
3  IADD
4  ICONST_3
5  POP
6  IRETURN
```

The analysis starts with the task of finding out which opcode provides the value 3 returned by IRETURN. In order to also deal with abstract values, we (also) use references to describe inputs and outputs of opcodes. Assuming that $iconst_3$ is the only reference used for the value 3 in this example, we use the pair $(\{iconst_3\}, \{In_6\})$ to denote that the origin of the reference $iconst_3$ is as relevant as the input of the IRETURN opcode. The next analyzed opcode, POP, also has an input, which also is the reference named $iconst_3$. Because of this we now have two items in the list of references to look for, namely $[\{(iconst_3\}, \{In_6\}), (\{iconst_3\}, \{In_5\})]$. Here, the most current entry is shown on the right and it contains the information that the last origin of $iconst_3$ (only) is as relevant as In_5, the input of POP.

When now analyzing ICONST_3 we not only see that this opcode provides a reference we are looking for, but we also can use the information from the list to see that the output of the ICONST_3 opcode is the input of the POP opcode. Because of this we add the implication $In_5 \rightarrow Out_4$ and remove the item $(\{iconst_3\}, \{In_5\})$ from the list. When considering IADD the remaining entry $(\{iconst_3\}, \{Rel_6\})$ in the list provides the information that its output is the input for IRETURN (i.e., we add $In_6 \rightarrow Out_3$). In total we can conclude that the code in lines 4 and 5 is not relevant for the returned value (if we initially only set Rel_6 to $\top$).

In order to also compute correct results for cases as in Example 4.5, we mimic the operand stack of JAVA BYTECODE by using a list of inputs. This list contains pairs of references and propositional variables encoding the relevance of the input. This way we can distinguish different usages of the same reference.

In the case of objects, the value is not only provided by passing around the reference, but also by writing to fields of the object. Also, changes to objects contained in fields of an object change its value. Similarly, arrays contain cells that may be changed by writing into them. Because of this we need to track write accesses in addition to the origin of the object or array.

Example 4.6 (Write Accesses to Objects) In this example, if the returned List object is marked as relevant, the origin of the object (the method invocation in line 4) must also be considered as relevant. Because the write accesses in lines 5 and 6 change the content of a relevant object, we also need to mark the corresponding opcodes as relevant. Furthermore, because x is written into list, for which we need to detect changes, changes to x also are relevant. This means that the write access in line 2 also is relevant for the value of list. In total, in this example all code must be considered as relevant.

```
1  List x = createList();
2  x.value = someNumber();
3
4  List list = createList();
5  list.value = someNumber();
6  list.next = x;
7
8  return list;
```

In order to achieve the desired effect, when analyzing the RETURN opcode we use the tuple $(\{\mathsf{list}\}, \{Rel_8\})$ to indicate that side effects to the returned reference list must be considered as relevant in case the user is interested in the returned value. With this information analysis of the write accesses in lines 5 and 6, which write to the reference mentioned in the tuple, leads to the implications $Rel_8 \rightarrow Rel_6$ and $Rel_8 \rightarrow Rel_5$. When considering the write access in line 6, we additionally consider the tuple $(\{\mathsf{x}\}, \{Rel_8\})$ when analyzing the preceding code. This leads to the encoding of

$$Rel_8 \rightarrow Rel_6 \wedge Rel_8 \rightarrow Rel_5 \wedge Rel_8 \rightarrow Rel_2.$$

In combination with the analysis of input values as described in the previous examples, in this example all lines of code must be marked as relevant for the returned value of list.

Making use of the Symbolic Execution Graph

The explanations mentioned so far do not differ much from known techniques for dead code analysis. However, in our analysis we do not work on the level of the individual opcodes, but instead consider states in the Symbolic Execution Graph. An immediate benefit is that the Symbolic Execution Graph already contains detailed information about the heap.

Example 4.7 (Sharing Information)

```
1  List list = createList();
2  List anotherList = something();
3  anotherList.value++;
4  return list;
```

If the relation between list and anotherList is not known, it must be assumed that the write access in line 3 also affects the value of list. Therefore, it would be unsafe to mark the code in lines 2 and 3 as irrelevant. However, if the Symbolic Execution Graph contains the information that list and anotherList do not share, it is safe to mark lines 2 and 3 as irrelevant for list.

When working on a Symbolic Execution Graph we can also make use of the fact that the graph can contain path-sensitive information. In the construction of the Symbolic Execution Graph we may have several states corresponding to a single opcode in the program. For example, when evaluating a conditional opcode, the Symbolic Execution Graph may contain two branches, where in each branch refined information corresponding to the evaluation of the condition is stored. This information is used in all following states of the Symbolic Execution Graph, so that the case analysis of subsequent branching opcodes may result in less cases.

Example 4.8 (Path-Sensitive Analysis)

```
1  List list = createList();
2  boolean isNull = (list == null);
3  int res = 0;
4  if (isNull && list != null) {
5    res++;
6  }
7  return res;
```

The code in line 5 is never executed because the corresponding condition is never fulfilled. The constructed Symbolic Execution Graph may contain two branches, one for the case that list is null, one for the case that list is not null. Therefore, we might also know that the branch condition is not satisfied. With our path-sensitive analysis the graph does not contain states for line 5. Because of that we do not add any implications connecting the return value with the opcodes corresponding to the code in line 5. As a consequence we can conclude that the code creating and using `list` is not relevant for the returned value.

4.3. Detailed Procedure

In this section we will formalize the ideas presented in Section 4.2. For this, in Section 4.3.1 we first consider programs without exceptions and method invocations and introduce the concepts of *tracking* data and *inputs*, which together are used to find all code that somehow creates or influences data that is marked as relevant. With these, we then present an algorithm computing this analysis. In Section 4.3.2 we provide a more in-depth analysis of how branches (including loops) can be handled to provide better results. Finally, in Sections 4.3.3 and 4.3.4 we explain how the technique can be extended to also analyze programs with method invocations and exceptions.

Preliminaries

During the analysis a propositional formula is constructed, which only consists of conjunctions of implications. We demand that each implication has the form $a \to b$ where a and b are propositional variables. As the constructed formula only contains such implications, finding the (minimal) model is straightforward by setting all variables to $\perp$. However, after the formula is constructed, the user may pick any subset of these variables and manually assign them to $\top$, for example to indicate that specific results of the computation are relevant. Because of the specific form of the formula finding a minimal model is straightforward by propagating the values, starting with the variables set to $\top$ by the user. Based on the user's choice of variables set to $\top$ a minimal model then gives information about parts of the program which can be considered irrelevant.

In the following definitions we will often work with states of the Symbolic Execution Graph and information provided in the topmost stack frame of a state (e.g., the current opcode). The following notation simplifies access to this information.

Definition 4.2 (Notation) If not stated otherwise, for a state s in the Symbolic Execution Graph which is not a program end, let $op(n)$ be the opcode of the topmost stack frame in s. Furthermore, let $ex(s)$ be the exception reference contained in the topmost stack frame of s. If the exception reference is not set, we define $ex(s) := \perp$.

Opcodes	**Example**	**Description**	**In**	**Out**
1–20, 168, 201	ICONST_0	constant value	0	✓
21–45	ILOAD_0	load from local variable	0	✓
54–78	ISTORE_0	store to local variable	1	×
46–53	IALOAD	load from array	2	✓
79–86	IASTORE	store to array	3	✓
87	POP	pop from operand stack	1	×
96 – 115, 120–131	IADD	binary arithmetic	2	✓
116–119	INEG	unary negation	1	✓
133–147	I2L	conversion	2	✓
148–152	LCMP	binary comparison	2	✓
153–158, 198–199	IFEQ	unary branching comparison	1	×
159–166	IF_ICMPEQ	binary branching comparison	2	×
170–171	TABLESWITCH	branch with several targets	1	×
172–176	IRETURN	return a value	1	✓
178	GETSTATIC	read from static field	0	✓
179	PUTSTATIC	write to static field	1	×
180	GETFIELD	read from instance field	1	✓
181	PUTFIELD	write to instance field	2	×
187	NEW	create new instance	0	✓
188–189	NEWARRAY	create new array	1	✓
190	ARRAYLENGTH	get length of array	1	✓
191	ATHROW	throw exception	1	×
192	CHECKCAST	check cast	1	×
193	INSTANCEOF	instance of	1	✓

Figure 4.3.: properties of most opcodes

4.3.1. Analysis of Code Without Exceptions and Method Invocations

To ease the presentation, we first restrict the analysis to code that never throws an exception and never invokes any method. After we presented the analysis for this simplified code, we remove the restrictions and adapt the technique accordingly.

In a first step, we define propositional variables and implications which are used as the basis of the formula.

Definition 4.4 (Initial Variables and Implications) Let N be the set of states in the Symbolic Execution Graph. For each state $n \in N$ we introduce variables and add implications to the formula as follows. If n is a program end, nothing is done. For each other state we introduce the variables Rel_n (used to encode the relevance of state n) and $Rel_{op(n)}$ (used to encode the relevance of opcode $op(n)$). We add the implication $Rel_n \to Rel_{op(n)}$ to indicate that $op(n)$ is relevant if any state with that opcode is relevant. This way, we only show the user the information that an opcode is irrelevant if *all* states corresponding to this opcode are not found to be relevant. In other words, parts of the program may be only relevant to a special case of the computation, which may correspond to one of many branches in the Symbolic Execution Graph. As such, to only show the desired result for such special cases, we need to consider all states for each opcode.

In Fig. 4.3 properties for 230 out of all 256 opcodes are shown. For those we can directly define which variables we need and which implications we add:

If, according to Fig. 4.3, $op(n)$ is an opcode with output, we introduce the variable $Out_{n,1}$ and add the implication $Out_{n,1} \to Rel_n$. If $op(n)$ has i inputs, we introduce the variables $In_{n,1}$ to $In_{n,i}$. Furthermore, we add the implications $Rel_n \to In_{n,1}$ to $Rel_n \to In_{n,i}$.

For the remaining 25 opcodes we need further explanations.

- The opcodes RET, NOP, IINC, GOTO, GOTO_W, RETURN, and WIDE do not have input or output values we want to encode using propositional variables, so nothing needs to be done.
- The opcodes BREAKPOINT, IMPDEP1, IMPDEP2, MONITORENTER, MONITOREXIT, and INVOKEDYNAMIC are not supported by this analysis.
- Handling of the invoke opcodes INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC, and INVOKEINTERFACE is explained in Section 4.3.3.

For the following opcodes we state how many inputs and outputs they have. For i inputs and j outputs we introduce the variables $In_{n,1}$ to $In_{n,i}$ and $Out_{n,1}$ to $Out_{n,j}$. We also add the implications $Rel_n \to In_{n,1}$ to $Rel_n \to In_{n,i}$ and $Out_{n,1} \to Rel_n$ to $Out_{n,j} \to Rel_n$.

- The opcode SWAP has 2 input and 2 outputs.
- The opcodes DUP, DUP_X1, and DUP_X2 each have 1 input and 2 outputs.
- The opcodes DUP2, DUP2_X1, and DUP2_X2 either have 1 input and 2 outputs or 2 inputs and 4 outputs, depending on the content of the operand stack. In the Symbolic Execution Graph it can directly be seen which of those two variants exists in a given state.
- The opcode POP2 can have 1 or 2 inputs, depending on the content of the operand stack. As in the case of DUP2, this information can be directly seen in the Symbolic Execution Graph.
- The opcode MULTIANEWARRAY has i inputs where i is a constant defined for each occurrence of this opcode. This constant can easily be seen in the Symbolic Execution Graph.

The variables introduced in Definition 4.4 need to be connected with those of other states so that the constructed formula also contains information about which parts of the program are influenced by other parts. In order to find out the necessary connections, we need to consider which influences can exist in a program.

Tracked Data and Inputs

In the analysis, we make use of two similar concepts to capture how data is modified and provided to other parts of the program. With *Inputs* we are able to find which preceding opcode provides the value used by some opcode. This was motivated in Example 4.5. By having *Tracked Data* we are able to detect write accesses as already motivated in Example 4.6. We also use this concept to find code that writes into local variables and static fields.

For references corresponding to object instances or arrays it may be necessary to find preceding opcodes providing the reference (for example by creating the object or loading it from some field) where, in addition to that, we are also interested in code that changes the contained data. In this case, we make use of both concepts.

Certain properties of referenced data cannot be changed. For example, the length of an array cannot be modified. Thus, for the analysis of ARRAYLENGTH opcodes we do not need to track changes (*Tracked Data*) to the array. However, we might need to find the origin of the data (*Inputs*).

Tracked Data

We first consider the case of **local variables**. An opcode reading from a local variable (e.g., ILOAD) provides the value that was last stored into the local variable. Because of this, we just need to remember which local variable we are interested in and then visit the states leading to the current state in reverse order. As soon as an opcode is found that defines the value of the local variable, we add the implication that the storing opcode (and, therefore, also its input) is as relevant as the input of the reading opcode.

In the case of **static fields** we have a very similar situation, where we consider opcodes reading from/writing into static fields instead of reading from/storing into local variables. The same idea can also be used to find opcodes that change **objects or arrays** for which such changes need to be found. As we already saw in Example 4.6, when looking for changes to r it does not suffice to find the last opcode that changes r. Instead, we need to consider all opcodes up to the point where r is created.

In Fig. 4.5 you can see all opcodes for which we need to track references, local variables, or static fields. In Example 4.5 we already motivated that we need to track references as data may be abstracted. The data may also be stored into and read from local variables or static fields, so we also define when to track local variables and static fields. As an example, if we are interested in where the value provided by an ILOAD_0 opcode comes from, we know that it was stored into the first local variable. Thus, we start tracking the local variable, which helps us finding the ISTORE_0 opcode providing the value we are interested in. Note that, for example, it is not necessary in the case of an ARRAYLENGTH opcode to detect changes to the array because the array length is immutable. Similarly, we do not need to track the objects investigated by the opcodes IFNULL or INSTANCEOF. In Fig. 4.6 we define which opcodes modify data that may be tracked for an opcode mentioned in Fig. 4.5.

Opcodes	**Example**	**Tracked**
21–45	ILOAD_0	local variable
46–53	IALOAD	reference of array
178	GETSTATIC	static field
180	GETFIELD	reference of object
132	IINC	local variable
169	RET	local variable

Figure 4.5.: data tracked for certain opcodes

Opcodes	**Example**	**Modified**
54–78	ISTORE_0	local variable
79–86	IASTORE	reference of array
179	PUTSTATIC	static field
181	PUTFIELD	reference of object
132	IINC	local variable

Figure 4.6.: tracked data changed by opcodes

In the algorithm presented later in this section we will use the variable *Track* that maps each state to a set of references, local variables, and static fields we need to track. In *Track* we not only store data for which we need to find code changing it (this can be a local variable, a static field, or a set of references). In addition we store the reason for tracking the data using propositional variables. Then, if in the analysis we find modifications to tracked data (e.g. a write access to a tracked reference), we can add the corresponding implication. The propositional variables need to be included in *Track*, as without this information we do not know *how* relevant changes to tracked data are.

As an example, we may track references $\{r_1, r_2\}$ and denote that opcodes modifying these references inherit the relevance encoded in proposition variables $\{Rel_n, Rel_m\}$. If we now change r_1 in an opcode for which we encode the relevance as Rel_p, we add the implications $Rel_n \to Rel_p$ and $Rel_m \to Rel_p$.

Inputs

As seen in Example 4.5, to find the origin of inputs for an opcode, we need to use a stack where each element contains the reference that is the input and the propositional variable which will be used to create the desired implication. Because of details explained later in this chapter, it does not suffice to consider only a single reference here, so we use a set of references instead of a single reference. Also, we use a set of variables instead of a single variable here.

It is quite straightforward to define the inputs of certain states, based on Definition 4.4. As an example, if we have an IADD opcode reading two references from the operand stack, we create inputs for those two references.

However, the list we use for a specific state must not only contain the corresponding inputs, but also the remaining inputs of all following opcodes need to be regarded.

Example 4.9 The bytecode for the expression `1 + 2` is shown in the first column of the following table. In the second column the operand stack is shown *before* execution of the corresponding opcode.

Code	**Operand stack**	**Inputs**
1 ICONST_1	ε	$[]$
2 ICONST_2	$iconst_1$	$[(\{iconst_1\}, \{In_{3,1}\})]$
3 IADD	$iconst_1, iconst_2$	$[(\{iconst_1\}, \{In_{3,1}\}), (\{iconst_2\}, \{In_{3,2}\})]$

While the opcodes in line 1 and 2 do not have any input, the opcode in line 3 has two inputs. We use a list of inputs $[(\{iconst_1\}, \{In_{3,1}\}), (\{iconst_2\}, \{In_{3,2}\})]$ for the addition in line 3. The opcode in line 2 creates the reference $iconst_2$ and, thus, we have found the origin of one of the inputs for the IADD opcode. The opcode in line 2 has no input, so in contrast to IADD the input list should be empty. However, as we still need to find the origin of $iconst_1$, which is relevant for the IADD opcode, we treat $iconst_1$ as an input to ICONST_2. When analyzing ICONST_1 we see that it provides the reference $iconst_1$ that is an input for the opcode in line 3. This leads to the desired implications $In_3 \rightarrow Out_2$ and $In_3 \rightarrow Out_1$.

To implement the idea hinted at in Example 4.9, we define the inputs list for a given state based on the inputs list of all direct successors in the graph. Here, the general idea is to have the entries of the successor states at the front of the list, while inputs needed for the current opcode are added at the end of the list. This way, when we analyze an opcode creating a reference contained in the list, the list contains the information of where the data is used and, thus, how relevant it is to provide it.

If the edge between two states n and m in the graph is an evaluation edge or an instance edge, then m is the only successor of n. Because of this, we can just append the list of inputs created for the opcode in n to the list representing the inputs of m.

In the case of a refine or split edge, we may have several successors $m_1, m_2, \ldots$ of a state n in the graph. Thus, we need to take care that when propagating information from m_i to n, no information is lost which was already propagated for some other $m_{i'}$. However, as all successors $m_1, m_2, \ldots$ share the same opcode, the corresponding inputs lists have the same length. Because of this it also is straightforward how to combine several lists of the successors into the start of the single list used for the current state.

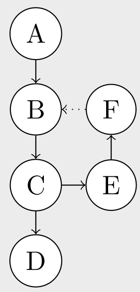
In the algorithm presented in the next subsection we use the variable *Inputs* that maps each state of the Symbolic Execution Graph to a list of entries as described above, each entry containing a set of references and a set of propositional variables.

Algorithm

We now present the algorithm that produces implications connecting the propositional variables encoding the relevance of the states, based on tracked data and inputs. Because of loops in the Symbolic Execution Graph we use a simple fixed-point algorithm.

Example 4.10 (Instance Field Only Read in Loop) Below, code containing a simple loop is shown. To the right a simplified version of a Symbolic Execution Graph is shown, where the edge from state F to state B closes a cycle using an instance edge (corresponding to the loop in lines 3–5).

```
1   int res = 0;
2   int incr = someValue();
3   for (int i = 0; i < 100; i++) {
4     res += incr;
5   }
6   return res;
```



When computing the relevance of the program, starting with the state D corresponding to the return of res, we may first propagate the information along the (reversed) path $D \leftarrow C \leftarrow B \leftarrow A$. During this computation there is no reason to add an implication that could cause the write access to incr in line 2 to be relevant. However, when now considering the edges $B \leftarrow F$ and $F \leftarrow E$ corresponding to the loop body, we see that we need to track writes to incr because the code in line 4 reads from that local variable and influences the value of res. As a consequence, we need to re-examine states C, B, and A with the updated information that writing to incr is relevant for res, leading to the result that, indeed, the method invocation in line 2 is relevant for the returned value.

The algorithm CreateFormula, shown in Algorithm 17, first initializes the formula with implications connecting the variables used for the outputs and inputs of a state with the variable encoding the relevance of a state (InitFormula in line 6, cf. Definition 4.4).

Before the actual computation starts, the user needs to define references for which changes are relevant and should be detected (*InitialTrack*). For example, the user may decide that for computations finishing in an end state n, where the only incoming evaluation edge corresponds to evaluation of RETURN r, changes to r should be tracked. This would lead to $\mathit{InitialTrack}(n) = \{(\{r\}, \{\mathit{Rel}_r\})\}$ where Rel_r is set to $\top$. Marking references using *InitialTrack* may only be done for end states. This information is then propagated in the graph, so that for all opcodes influencing the value corresponding implications are added. Without this information, as the program possibly does not contain an opcode reading from these references, the analysis would not have any reason to find changes to the referenced objects or arrays and, thus, would not add implications that could cause the corresponding parts of the program to be marked as relevant.

Starting with the end states of the graph, for a state n the information of a successor state m is propagated to n. This is ensured by first dealing with the end states (for which no successors exist) in lines 9–12. Then, in line 14, $m \notin \mathit{Todo}$ at first only holds for end states. Thus, analysis starts with the edges leading to end states, for example from states containing RETURN opcodes.

In the case of an evaluation edge, we use Eval (Algorithm 18). In the case of refine, split, or instance edges, we use RefineSplitInstance (Algorithm 19). Both algorithms modify the variables *Track* and *Inputs*, containing the information we need to propagate. Furthermore, they add implications to the formula φ and add predecessor states to *Todo*, if they need to be re-analyzed. This was hinted at in Example 4.10. The following explanations of the individual algorithms give further details.

Evaluation Edges

In Algorithm 18 we deal with the case of two states n and m, where the edge (n, m) is an evaluation edge. In lines 3–4 we copy the *Inputs* list and *Track* set of the successor state m, taking care of possible renamings. In the case of *Track* we also remove entries that are non-existent in n using GC (for Garbage Collection). This can happen, for example, when writes to a reference are tracked where the corresponding object is not yet created in n.

In lines 5–9 we deal with the case that in n an output is created for which an entry in *Inputs* exists. If this is the case, we add implications to φ, indicating that state n is as relevant as the inputs it provides. Here, CreatedOutputs returns the references and the corresponding variables of the output, if any, according to Definition 4.4. As motivated in Example 4.5, we need to find the corresponding entry in the *Inputs* list, even if *CreatedOutput* is contained more than once. Thus, the function Rightmost-

Algorithm 17: CREATEFORMULA

Input: Symbolic Execution Graph $G = (N, E, \mathcal{L})$, *InitialTrack*
Output: propositional formula φ encoding the relevance of states

1: $\mathit{Todo} = N$
2: $\mathit{Track} = \emptyset$
3: $\mathit{Inputs} = \emptyset$
4:
5: // initial implications according to Definition 4.4
6: $\varphi = \textsc{InitFormula}(G)$
7:
8: // initialize end states
9: **for all** $n \in \textsc{EndStates}(N)$ **do**
10: $\quad\mathit{Track}(n) = \mathit{InitialTrack}(n)$
11: $\quad\mathit{Inputs}(n) = \emptyset$
12: $\quad\mathit{Todo} = \mathit{Todo} \setminus \{n\}$
13:
14: **while** $\exists n \in \mathit{Todo} \wedge (n, m) \in E \wedge m \notin \mathit{Todo}$: **do**
15: $\quad$**if** $\mathcal{L}(n, m) = \textsc{Eval}$ **then**
16: $\quad\quad$// propagate information for evaluation states
17: $\quad\quad\textsc{Eval}(\mathit{Todo}, \mathit{Track}, \mathit{Inputs}, \varphi, n, m)$
18: $\quad$**else if** $\mathcal{L}(n, m) \in \{\textsc{Refine}, \textsc{Split}, \textsc{Instance}\}$ **then**
19: $\quad\quad$// combine information for refine, split, and instance edges
20: $\quad\quad\textsc{RefineSplitInstance}(\mathit{Todo}, \mathit{Track}, \mathit{Inputs}, n, m)$
21:
22: $\quad\mathit{Todo} = \mathit{Todo} \setminus \{n\}$
23:
24: **return** φ

VARIABLES returns the propositional variables of the rightmost entry in $\mathit{Inputs}(n)$ which contains *CreatedOutput*. The function REMOVERIGHTMOST then removes this entry from $\mathit{Inputs}(n)$.

In lines 10–18 we add implications for each tracked local variable, static field, or reference which is modified by n. Detecting modifications of local variables and static fields is quite straightforward, as the opcodes mentioned in Fig. 4.6 directly name the corresponding variable or field. However, to detect changes to an object we do not only need to find direct writes to the references mentioned in *Track*, but we also need to take sharing on the heap into account. For example, if we have a list of the form $\mathsf{a} \to \mathsf{b} \to \mathsf{c}$ and the value inside the list element c is changed, this also modifies data visible from the first list element a. Because of that we define that a reference r in state s is modified if $r \leadsto r'$ (cf. Definition 1.42) and r' is the direct target of a write access. The modified references are computed using WRITTEN as shown in line 15.

If any reference r is modified, we start tracking the reference that is stored into r (line 15). In the case of a modified object or array we continue tracking it, as a single

Algorithm 18: EVAL

Input: *Todo*, *Track*, *Inputs*, φ as in Algorithm 17, states n and m

Output: modifications to *Todo*, *Track*, *Inputs*, φ

1: $\mathit{InputsOld} = \mathit{Inputs}(n)$
2: $\mathit{TrackOld} = \mathit{Track}(n)$
3: $\mathit{Inputs}(n) = \textsc{RenameInputs}(\mathit{Inputs}(m), n, m)$
4: $\mathit{Track}(n) = \mathrm{GC}(\textsc{RenameTrack}(\mathit{Track}(m), n, m), n)$
5: **for all** $(\mathit{CreatedOutput}, \mathit{Out}_{n,i}) \in \textsc{CreatedOutputs}(n)$ **do**
6:     **if** $\mathit{InVars} = \textsc{RightmostVariables}(\mathit{Inputs}(n), \mathit{CreatedOutput})$ **then**
7:         // n creates an output we are looking for
8:         $\mathit{Inputs}(n) = \textsc{RemoveRightmost}(\mathit{Inputs}(n), \mathit{CreatedOutput})$
9:         $\varphi = \varphi \wedge \bigwedge_{\mathit{In} \in \mathit{InVars}} \mathit{In} \to \mathit{Out}_{n,i}$
10: **for all** $(T, \mathit{Vars}) \in \mathit{Track}(n) \wedge T$ is modified by n **do**
11:     // n changes T which is tracked
12:     $\varphi = \varphi \wedge \bigwedge_{\mathit{Var} \in \mathit{Vars}} \mathit{Var} \to \mathit{Out}_{n,i}$
13:     **if** T is a reference **then**
14:         // also track the source
15:         $\mathit{Track}(n) = \mathit{Track} \cup \{(\textsc{Written}(n), \mathit{Vars})\}$
16:     **else**
17:         // stop tracking local variable or static field
18:         $\mathit{Track} = \mathit{Track} \setminus \{(T, \mathit{Vars})\}$
19: $\mathit{Inputs}(n) = \mathit{Inputs}(n) \cup \textsc{NewInputs}(n)$
20: $\mathit{Track}(n) = \mathit{Track}(n) \cup \textsc{NewTrack}(n)$
21: **if** $\mathit{Inputs}(n) \neq \mathit{InputsOld} \vee \mathit{Track}(n) \neq \mathit{TrackOld}$ **then**
22:     $\mathit{Todo} = \mathit{Todo} \cup \textsc{Predecessors}(n)$

object instance or array may be changed by an arbitrary number of preceding opcodes. Instead, any tracked local variable or static field is only written once and we do not need to find further opcodes writing into the variable or field. Thus, the corresponding entry is removed. Using the functions NEWINPUTS and NEWTRACK we find out the direct inputs of n and new data we need to track. These are added to *Inputs* and *Tracks* so that, when analyzing preceding states, we can see which states influence the behavior of n. The actual content of NEWINPUTS and NEWTRACK is built based on Definition 4.4 and Fig. 4.6. Finally, in lines 21–22 we check if the information in *Track* or *Inputs* differs from a previous run of EVAL on (n, m). If this is the case, we need to (re)consider all predecessor states with the updated information.

Refine, Split and Instance Edges

In Algorithm 19 we deal with states n and m where (n, m) is a refine, split, or instance edge. While there can only be a single outgoing evaluation edge for each state, there may be several outgoing edges of n. Because of that, for evaluation edges, we re-computed the information we need to store for a state solely based on the information we have for the unique successor state. However, in the case of refine or split edges the information we store for a state depends on the values of all successor states. This algorithm is also used for instance edges. While there can be no other outgoing edge of n if (n, m) is an instance edge, no computation is done from n to m. Therefore, we use the same renaming techniques presented here also for instance edges.

In the case that we do not have any information for n, we just copy the information of the successor state (lines 4–5 and 8–9). Here, as in Algorithm 18, we take care of possible renaming and remove entries that do not exist, yet. Because of equality refinement it can happen that a single reference r in m corresponds to two references r', r'' with $r' =^? r''$ in n. In this case the set $\{r', r''\}$ is used in n where the set containing r was used in m. Furthermore, due to instance refinements it can happen that in m the heap contains the information $x.f = r$ (indicating that some object x contains the reference r in its field f) while in n this information does not exist. To solve this problem, RenameTrack adds the reference of the predecessor object (which is refined from n to m) in place of r.

If we already have information for state n (because we already dealt with any successor), we must not forget this information. Instead, we combine the information from the state m with the already existing information. This is done in lines 7 and 11. In the case of *Inputs* (line 7) we make use of the fact that the lists of n and m have the same length. Now, we just combine the corresponding entries in the list by considering the union of the references and variables components, respectively. The variable *Track* contains a set of tuples for n, where the first component is a set of references, a local variable, or a static field. The second component is a set of propositional variables. MergeTrack combines the entries for local variables and static fields by just taking the union of the variables. In the case of references, corresponding entries (taking possible renaming and refinement into account) also are identified and the union of variables is taken. As in Algorithm 18 we (re)add all predecessor states to *Todo* if we updated any information for the state (in lines 12–13).

4.3.2. Branches

The procedure presented so far is incomplete in the sense that for states of branching opcodes no implications are added. As a consequence these opcodes and also the inputs that determine the branching behavior are not considered to be relevant. To correct this, we need to define when a branch is relevant and how to detect this.

Example 4.11 (Relevant Branch) In this example the opcode incrementing `res` clearly is relevant, if we assume that the returned value is relevant. Depending on the value of `x` line 4 is skipped, so the branch in line 3 and its input `x` must also be considered to be relevant.

Algorithm 19: RefineSplitInstance

Input: *Todo*, *Track*, *Inputs* as in Algorithm 17, states n and m
Output: modifications to *Todo*, *Track*, *Inputs*

$\mathit{InputsOld} = \mathit{Inputs}(n)$
$\mathit{TrackOld} = \mathit{Track}(n)$

if $\mathit{Inputs}(n)$ is undefined **then**
  $\mathit{Inputs}(n) = \textsc{RenameInputs}(\mathit{Inputs}(m), n, m)$
else
  $\mathit{Inputs}(n) = \textsc{MergeInputs}(\mathit{Inputs}(n), \textsc{RenameInputs}(\mathit{Inputs}(m), n, m))$
if $\mathit{Track}(n)$ is undefined **then**
  $\mathit{Track}(n) = \mathrm{GC}(\textsc{RenameTrack}(\mathit{Track}(m), n, m), n)$
else
  $\mathit{Track}(n) =$
  $\textsc{MergeTrack}(\mathit{Track}(n), \mathrm{GC}(\textsc{RenameTrack}(\mathit{Track}(m), n, m), n))$
if $\mathit{Inputs}(n) \neq \mathit{InputsOld} \vee \mathit{Track}(n) \neq \mathit{TrackOld}$ **then**
  $\mathit{Todo} = \mathit{Todo} \cup \textsc{Predecessors}(n)$

```
1  int res = 0;
2  int x = someNumber();
3  if (x > 0) {
4    res++;
5  }
6  return res;
```

The key idea that can already be seen in this simple example is that a branch is relevant if, by branching to a certain target, a relevant opcode may be skipped. In other words, because the value of res is relevant, it is important to know whether the increment operation in line 4 is executed or not.

Even if all possible branch targets contain relevant code, the choice which of these is executed (which also means, which of those is *not* executed) makes the branching decision relevant. In the example above, we could add another branch target like else res--. Then both res++ and res-- are relevant. As it is important to know *which* of those statements is executed, also the branch condition is relevant.

Theoretically it is possible that the code in *all* branch targets is equivalent. In these cases our analysis would be less precise than possible. However, as these cases are pathological and determining equivalence of arbitrary code is an undecidable problem, we just accept this imprecision.

To better understand the concept of a relevant branch, we also need to think about irrelevant branches. Instead of defining a branch as relevant if its branching behavior influences the series of opcodes that is executed (which is true for virtually all branches), we take the relevance of the code reachable in the individual branch targets into account.

Example 4.12 (Irrelevant Branch) Depending on the value of `x` the code in line 4 or line 6 is skipped. Thus, if the value of `x` is relevant, also the branch condition is relevant.

```
1 int res = 0;
2 int x = someNumber();
3 if (x > 0) {
4   x++;
5 } else {
6   x--;
7 }
8 return res;
```

However, if we only consider the returned value to be relevant, we do not introduce any corresponding implication that could mark the branch as relevant.

Loops

Loops in a Java program are implemented in Java Bytecode using branches that either branch to the loop body or skip it. At the end of the loop body an unconditional (non-branching) jump leads back to code computing the inputs for the branch of the loop condition.

In order to be able to identify irrelevant branches that correspond to loops, we must re-visit the idea of skipping code. Clearly, if by taking a branch some relevant code is skipped, the branch must be considered as relevant. However, in a loop we have a different situation. Here the relevant code can be skipped by executing the loop body, but it might be executed eventually when the loop terminates.

Example 4.13 (Loop Relevance) In each traversal of the loop, the relevant code in line 6 is skipped. However, as soon as the loop finishes, it will be executed. In other words, on each execution path leading to the code in line 6 (either by traversing the loop at least once but only finitely often or by skipping it entirely), the loop only determines how many irrelevant computations are done before the relevant computation in line 6 is executed.

```
1 int res = 0;
2 int x = getRandomNumber();
3 while (x > 0) {
4   x = getRandomNumber();
5 }
6 res++;
7 return res;
```

It is easy to see that the loop does not influence the value of `res`. If it is terminating, we clearly should mark it as irrelevant. However, we also need to consider the case that the loop does not terminate. Then the computation starting in line 6 never is executed. However, a non-terminating loop can be seen as a different kind of bug. There already exist techniques to find non-terminating loops (or prove that all loops are terminating). Because of that we delegate the task of proving termination to the user and may mark loops as irrelevant even though they may be non-terminating – which is what an optimizing compiler removing dead code must not do.

The ideas presented in the previous examples are now combined into a definition of when a branching opcode must be considered as relevant. First, we need to define the function $\mathcal{R}$ that computes the reachable opcodes inside the same method.

Definition 4.7 (Reachable Opcodes) Let $\mathcal{G} = (N, E, \mathcal{L})$ be a Symbolic Execution Graph. Let o be an opcode in method m. Then we define $\mathcal{R}(o)$ as the minimal set with

- $\forall n \in N : op(n) = o \Leftrightarrow o \in \mathcal{R}(o)$
- $\forall (n, m) \in E : \mathcal{R}(op(m)) \subseteq \mathcal{R}(op(n))$

Definition 4.8 (Branch Relevance) Let o_b be a branching opcode. Let $o_1, \ldots, o_n$ (with $n > 1$) be the branching targets of o_b. Then we define o_b to be relevant if there exists a branch target o_i and an opcode $o \in \mathcal{R}(o_b)$ where $o \notin \mathcal{R}(o_i)$ and o is relevant.

Following this definition, a loop can only be relevant if it contains relevant code in the loop body (because the loop body may be skipped). However, relevant code executed after the loop terminates (e.g., line 6 in Example 4.13) does not cause the loop to be relevant (because, even when traversing the loop, it may still be executed).

In Definition 4.8 we assume that the relevance of every single opcode is known. However, this is not the case in this analysis. Instead of directly marking a branch as relevant, we provide implications that cause the branch to be relevant if certain opcodes are relevant.

Definition 4.9 (Implications Encoding Branch Relevance) Let $o_b, o_1, \ldots, o_n$ as in Definition 4.8. Then we add the implications

$$\{\mathit{Rel}_o \rightarrow \mathit{Rel}_{o_b} \mid o \in \mathcal{R}(o_b) \wedge o \notin \mathcal{R}(o_i) \wedge 1 \leq i \leq n\}$$

4.3.3. Method Invocations

When a method is invoked, the arguments of the method are provided on the operand stack. Then, after evaluation of the invoke opcode, the arguments are removed from the operand stack and a new stack frame is put on top of the one containing the invoke opcode. In that new frame the arguments previously stored on the operand stack are now stored as local variables.

First consider an edge (n, m) where n contains a method invocation and m is at the start of the invoked method. If in the analysis we are interested in the contents of a local variable in m, we now need to adapt the analysis as the contents of the local variable in m may correspond to an argument provided on the operand stack in n. Thus, if we have an entry in *Track* corresponding to a local variable in m, we need to use this entry to create a corresponding *Inputs* entry in n. To take care of this special case, the functions RENAMEINPUTS and MERGETRACK used in EVAL need to be changed accordingly.

When returning from a method without return value, the topmost stack frame is dropped and the opcode of the stack frame below that is advanced. If a value is returned, when evaluating the return opcode the (only) reference on the operand stack is put onto the operand stack of the stack frame below. By defining that the corresponding return opcodes both have a single input value and also have a single value as output, the presented algorithm already produces the intended implications.

Using these changes we can integrate invoke and return opcodes into the already presented algorithm so that the origin of data can be followed through method invocations. However, so far we did not define when a method invocation is relevant. Here, the intuition is that a method invocation is relevant if any relevant code is executed. Therefore, we just need to consider all opcodes in the invoked method and add implications for each so that the relevance information is propagated to the invoke opcode.

Definition 4.10 (Method Invocation Relevance) Let n be a state with opcode o invoking a method. Let m be the evaluation successor of n where o' is the current opcode (which is the first opcode of the invoked method). Then we add implications

$$\{Rel_x \to Rel_o \mid x \in \mathcal{R}'(o')\}$$

Here, $\mathcal{R}'$ is defined like $\mathcal{R}$, but just considers opcodes in the same method as o'.

Extension of this approach to also handle recursive methods as in Chapter 3 is not part of this thesis and instead is left for future work.

4.3.4. Exceptions

If an opcode throws an exception, this exception may be handled by dedicated code (corresponding to code in a catch clause of JAVA). If no such handler exists, the current stack frame is removed from the call stack and evaluation continues as if the invoke opcode (which now is at the top of the call stack) threw the exception.

We start with the situation that an exception is thrown when evaluating state n, for example because of a division by zero. The successor state m is a copy of n, where the operand stack is empty and an exception reference is noted in the topmost stack frame. For most opcodes that can throw an exception, there also exists the possibility of standard evaluation (in fact, only ATHROW always throws an exception). Because of this we need to consider most opcodes that may throw a caught exception as branching opcodes, where one branch corresponds to standard evaluation and the other branch corresponds to throwing an exception which is caught in another part of the method. Similarly, if an opcode throws an exception which is not caught, relevant parts of the method that correspond to evaluation without throwing the exception may be skipped.

Example 4.14 (Exception) If $\mathsf{b} = 0$ an exception is thrown and the method abruptly terminates. However, in this case also the code in line 4 is not executed. Therefore, assuming that the side effect on someObject is relevant, also the otherwise irrelevant computation in line 3 is relevant, as it defines whether the exception is thrown or not.

```
1  int a = someNumber();
2  int b = someNumber();
3  int tmp = a/b;
4  someObject.value++;
5  return;
```

Because the internal exception handling process only works depending on the type of the exception, we do not need to track any changes to the exception reference. Therefore, we define $\textsc{NewTrack}(m) = \emptyset$ for all states m with $ex(m) \neq \perp$ (cf. Definition 4.2). We extend the definition of $\textsc{CreatedOutputs}$ by defining $\textsc{CreatedOutputs}(n) = \{(\{e\}, \{Rel_n\})\}$ for states n that throw an exception e. This way, if in m the exception reference e is an input, state n is marked as relevant as this input. In the evaluation of n to m the operand stack is emptied. If the operand stack in n contains j references, this evaluation corresponds to evaluating POP j times. Thus, we treat the dropped references as inputs to the (implicit) POP operations:

$$\textsc{NewInputs}(m) = [(\{r_1\}, \emptyset), \ldots, (\{r_j\}, \emptyset)]$$

These entries correspond to the reference on the operand stack of the topmost stack frame which are removed, but which are not considered to be inputs for n. Since these are dropped without influencing succeeding code, we use the empty variable set. With these modifications we can deal with edges (n, m) where evaluation of n throws an exception.

In order to detect if by throwing an exception relevant code may be skipped, we add implications as in Definition 4.9. If the graph contains the information that the exception is caught, we just need to consider the opcode handlers as additional branching targets of the opcode. If it is possible that the exception is uncaught we define an additional dummy branching target $\lightning$ with $\mathcal{R}(\lightning) := \emptyset$. This way, we add implications for all opcodes that can be skipped if an exception is thrown.

Finally, we consider the case of how a caught exception is handled after it is thrown. Depending on the type of the exception set in the topmost stack frame, the opcode is changed to the code handling the exception. Additionally, in the topmost stack frame the exception component is unset and the exception reference is placed on the operand stack (as the only element). For states n where the opcode is changed to the handler opcode and the set exception is moved to the operand stack, we just define $\textsc{NewInputs}(n) = \textsc{CreatedOutputs}(n) = \emptyset$. With these modifications the algorithm already presented also handles the case that an exception is caught.

4.4. Computing Results

As already mentioned, this analysis computes a propositional formula φ that can be used to identify code that is irrelevant for some intended result as defined by the user. A very natural choice is to mark the returned value of some method in addition to the exceptions leading to crashes as relevant.

With this choice, where the propositional variables corresponding to the parts defined as relevant are set to $\top$, a minimal model of φ is computed. With this model we can then identify irrelevant opcodes by checking if the corresponding variables are set to $\perp$.

However, just showing the user all irrelevant opcodes is not very helpful. For example, the opcode POP which removes an entry from the operand stack never is relevant. This opcode is used, for example, in the case of invoking a method with return value where this return value is not used in the program. A common example is the method java.util.Collection.remove(Object o) which provides a boolean result indicating if the argument was removed from the collection. When just calling remove without checking the returned value, the superfluous boolean value is removed from the operand stack using POP. So, instead of letting the user inspect all POP opcodes in the program, it would be better to ignore occurrences of POP in the result. This idea can be extended to other opcodes, all of which are not used for real computations designed by the programmer: GOTO_W, JSR, JDR_W, RET, WIDE, POP, POP2, DUP, DUP_X1, DUP_X2, DUP2, DUP2_X1, DUP2_X2, SWAP, NOP, RETURN (without return value). For all these opcodes we always define the relevance to be *neutral*.

In the case of a NEW opcode, we may trigger initialization of a class (which, in turn, may call arbitrary code). Thus, if this side effect is relevant, also the NEW opcode is marked as relevant. However, we never mark a NEW opcode as irrelevant, as this opcode also does not correspond to a real computation designed by the programmer (thus, NEW either is relevant or neutral). Instead, invoking a constructor on the created object or storing the created object into a variable may be irrelevant.

When compiling Java to Java Bytecode for every created object also a constructor of each super class, up to java.lang.Object, is invoked. In the case that the constructors do nothing relevant, the corresponding INVOKESPECIAL opcodes are marked as irrelevant. However, for the programmer it is not possible to create an object without this invocation. Therefore, we detect which INVOKESPECIAL opcode marked as irrelevant can be considered relevant if each call to the constructor of java.lang.Object (which does nothing observable outside the Java Virtual Machine) is marked as relevant. These invoke opcodes (and all ALOAD_0 opcode immediately preceding them) then are shown as neutral code to the user, so that it is easier to concentrate on real bugs.

With these changes, every opcode of the program can be marked as relevant, neutral, or irrelevant. Neutral opcodes may be used as part of relevant computations, but also as part of irrelevant computations. So, when showing the results of the analysis to the user, neutral opcodes are ignored. When now considering large parts of the program marked as irrelevant (with an arbitrary number of neutral opcodes in between), showing these to the user can help finding bugs.

4.5. Optimizations

In our experiments we found some minor tweaks that can be implemented in order to get better results. A simple example is the negation using INEG. The Symbolic Execution Graph may contain the information that the argument is 0. Assuming state n corresponds to this negation, by replacing the implications $Out_{n,1} \to Rel_n$ and $Rel_n \to In_{n,1}$ with $Out_{n,1} \to In_{n,1}$ we would transfer the relevance of the result to the (identical) input without marking the negation itself as relevant. Thus, with this optimization it is possible to find useless negations in the code.

Similar ideas can be used for other arithmetic operations like multiplication with 1, storing reference x into a local variable already containing x, or executing assignments $x.f = y$ where the content of field f already is y.

4.6. Conjecture

In this section we develop a conjecture with the goal to formally state the idea behind irrelevant code. In most cases irrelevant code can be removed without influencing the result of the computation. For example one could remove any INEG opcode without altering the semantics of the program if it is ensured that the created output is identical to the provided input. In more complicated situations this is not possible. Instead the irrelevant code must remain or be replaced by other opcodes so that the resulting program is still valid (and verifiable) Java Bytecode, as hinted at in the following example.

Example 4.15 (Irrelevant Return Value) The method set returns a boolean value indicating if the content of the field was changed or not.

```
1  public class Test {
2    Object f;
3
4    public void main(String[] args) {
5      Test t = new Test();
6      t.set(args);
7    }
8    private boolean set(Object x) {
9      Object old = this.f;
10     this.f = x;
11     return x != old;
12   }
13 }
```

The (only) invocation in line 6 does not make use of this value. Therefore, the computation in lines 9 and 11 is irrelevant. The corresponding opcodes cannot be removed, since the method set needs to return a boolean value. In the following code the computation in line 11 is replaced by a constant. Since the returned value is not used, this program is equivalent.

```
1  public class Test {
2    Object f;
3
4    public void main(String[] args) {
5      Test t = new Test();
6      t.set(args);
7    }
8    private boolean set(Object x) {
9      this.f = x;
10     return true;
11   }
12 }
```

One possibility might also be to change the signature of the set method so that the return type is void. Then, however, one would also need to replace all invoking opcodes, as these reference the signature of the invoked method.

Similar to the previous example, opcodes which provide an irrelevant computation result, but sometimes throw an exception, cannot be removed from the program since the exception sometimes needs to be thrown.

Example 4.16 (Irrelevant Opcode with Exception) Assume the returned value is relevant. The result of the division 1/x is not relevant. However, the opcode computing the division may throw an exception (if x is 0). Throwing this exception is relevant because it leads to res++, influencing the returned value which is marked as relevant.

```
int res = 0;
int x = someNumber();
try {
  int y = 1/x;
  x++;
} catch (ArithmeticException e) {
  res++;
}
return res;
```

By replacing the division (which produces an irrelevant output) by code that throws the same exception in the same situations it is possible to simplify the code.

```
int res = 0;
int x = someNumber();
try {
  if (x == 0) {
    throw new ArithmeticException();
  }
} catch (ArithmeticException e) {
  res++;
}
return res;
```

Thus, one cannot simply remove all opcodes marked as irrelevant, and obtain an equivalent program. However, allowing the removal of irrelevant opcodes is the core result of this analysis. Instead of giving (complicated) details of how to transform a program accordingly, we leave this to future work and just define how the programs constructed using the information of which code is irrelevant or relevant may look like.

Conjecture 4.17 Consider a program $\mathcal{P}$ for which we computed the irrelevant opcodes using the technique explained in this chapter.

A program $\mathcal{P}'$ is created by replacing only irrelevant opcodes in $\mathcal{P}$ by arbitrary code such that $\mathcal{P}'$ is verified bytecode [Ler03] and this analysis, when run on $\mathcal{P}'$ using an initial relevance information corresponding to that specified for $\mathcal{P}$, identifies all the replacement opcodes as irrelevant.

Consider a finite computation sequence in $\mathcal{P}$, starting in a state s. Let $v_{\mathcal{P}}$ be a value computed in this sequence which is marked as relevant by the user. Then, for all programs $\mathcal{P}'$ as described above, the computation sequence in $\mathcal{P}'$, starting in s, computes a corresponding value $v_{\mathcal{P}'}$ which is identical to $v_{\mathcal{P}}$.

The main conclusion from Conjecture 4.17 is that programs that result out of just removing irrelevant opcodes, where the resulting program still is valid bytecode, compute the same results (if these are marked as relevant by the user).

4.7. Demonstration

We implemented the technique presented in this chapter and experimented on small, artificial programs which trigger certain parts of the program to be irrelevant. These experiments confirmed that the technique indeed works. However, mainly because obtaining

Symbolic Execution Graphs for real programs is a future goal, we did not find actual bugs in real-world programs, yet.

In Example 4.1 a small program was presented which contains a simple bug. On the following pages you can see un-abbreviated code for this example, where the analysis is indeed able to indicate irrelevant code as explained earlier.

Example 4.18 This example shows the complete code for Example 4.1.

```
import java.util.Iterator;

public class Graph {
  public List nodes;

  public static void main(String[] args) {
    Random.args = args;

    Graph graphOne = createGraph();
    Graph graphTwo = createGraph();

    Node source = graphOne.getRootNode();
    for (Node node : graphTwo.getNodes()) {
      if (Node.areConnected(source, node)) {
        graphOne.addNode(node);
      }
    }

    Graph res = graphOne;
  }

  public static Graph createGraph() {
    int num = Random.random();
    Graph res = new Graph();
    for (int i = 0; i < num; i++) {
      Node node = new Node(i);
      res.addNode(node);
    }

    int max = Random.random();
    for (int i = 0; i < max; i++) {
      Node start = res.getNode(Random.random());
      Node end = res.getNode(Random.random());
      if (start != null && end != null) {
```

```
        start.addEdge(end);
      }
    }

    return res;
  }

  public Node getNode(int id) {
    List cur = nodes;
    while (cur != null) {
      if (cur.content.id == id) {
        return cur.content;
      }
      cur = cur.next;
    }
    return null;
  }

  public Node getRootNode() {
    return this.nodes.content;
  }

  public List getNodes() {
    return nodes;
  }

  public void addNode(Node node) {
    this.nodes = new List(node, this.nodes);
  }
}

class List implements Iterable<Node> {
  Node content;
  List next;

  public List(Node c, List n) {
    this.content = c;
    this.next = n;
  }

  public Iterator<Node> iterator() {
```

```
    return new Iterator<Node>() {
      Node cur = null;
      List next = List.this;

      public boolean hasNext() {
        if (this.next != null) {
          this.cur = next.content;
          return true;
        }
        return false;
      }

      public Node next() {
        this.next = this.next.next;
        return this.cur;
      }

      public void remove() {
      }
    };
  }
}

class Node {
  public int id;
  List out;

  public Node(int i) {
    this.id = i;
  }

  public void addEdge(Node end) {
    this.out = new List(end, this.out);
  }

  public static boolean areConnected(Node node, Node other) {
    for (Node n : node.out) {
      if (n == other) {
        return true;
      }
    }
```

```
117      return false;
118    }
119  }
120
121  class Random {
122    static int count;
123    static String[] args;
124
125    public static int random() {
126      int res = args[count].length();
127      count++;
128      return res;
129    }
130  }
```

In Fig. 4.11 the control flow graph of the main method is shown, where opcodes analyzed to be irrelevant are shown in a diamond shape. Relevant opcodes have a rectangle shape, and neutral opcodes (an unconditional JMP and the RETURN opcode) are shown as ovals. A colored variant of this representation is computed automatically, the user does not have to provide any more input than the program to analyze.

Using the presented analysis, one could simplify the code according to Conjecture 4.17 and obtain the following code. As you can see, the programmer then can easily see that his intentions are not met, and the bug can be found.

Example 4.19 When marking res in line 19 as relevant and removing code from Example 4.18 identified as irrelevant using this analysis, the main method can be simplified as follows:

```
public static void main(String[] args) {
  Random.args = args;

  Graph graphOne = createGraph();
  createGraph();

  Graph res = graphOne;
}
```

4.8. Conclusion and Outlook

We presented a technique which makes use of the information available in the Symbolic Execution Graph created for a specific program. By analyzing definitions and usages of values in the program based on the information available in the Symbolic Execution Graph and a subsequent constraint-based analysis, it is possible to point the programmer to code which does not contribute to the intended results. For this, the analysis makes use of the inter procedural and context sensitive nature of the analysis presented in Chapter 1.

Thus, in addition to showing possible bugs like throwing NullPointerExceptions or non-termination as demonstrated in [BSOG12] and proving termination, this demonstrates that Symbolic Execution Graphs can also easily be used for further analyses.

In a next step, the ideas of this chapter should be formalized and shown correct. For this, it is necessary to properly define the desired outcome of the analysis. As already hinted at in Section 4.6, directly removing irrelevant code is not possible in all cases. Showing which code is found to be irrelevant to the user, for example by showing a colored control flow graph, is not hard to accomplish. However, in order to prove correctness of this approach, first a program transformation resulting out of this technique needs to be defined.

Furthermore, a graphical user interface where the user (programmer) can define the inputs to the algorithm (which opcodes and references are relevant) should be created. This information could also be provided by adding Java annotations to the program (for example @relevant). The results of the analysis should be presented in a practical way, for example by directly highlighting the analyzed code.

Finally, if one likes to extend this technique to find bugs in real-world programs, it is necessary to create Symbolic Execution Graphs reasonably fast. Thus, we refer to the conclusion of Chapters 1 and 3.

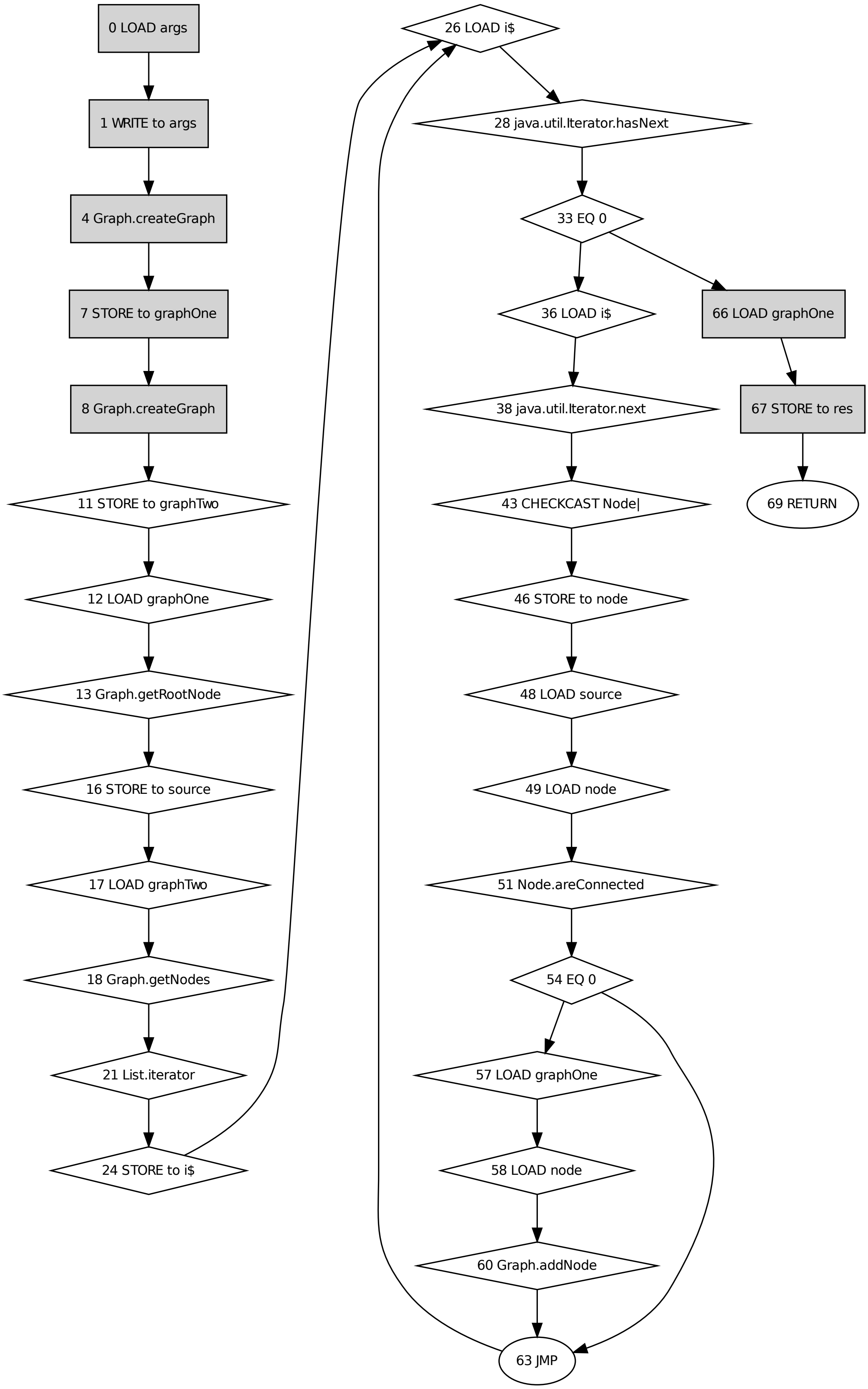


Figure 4.11.: Result of analysis

Conclusion

In this thesis we developed a new transformation from imperative programs written in Java Bytecode to Symbolic Execution Graphs. One application of these graphs is termination analysis of the original programs, as discussed in detail in the PhD thesis of Marc Brockschmidt [Bro14].

In Chapter 1, we presented a technique based on symbolic execution with abstraction to transform any non-recursive Java Bytecode program into a Symbolic Execution Graph. Using refinement and a quite precise abstraction the information contained in the individual states is very detailed. In comparison, the tools COSTA [AAC+08] and Julia [SMP10] use path length abstraction, yielding less precise results. In order to obtain this precise abstraction, we use several heap predicates to describe connections on the heap. Furthermore, we have shown how, using refinement, evaluation is possible even if critical information is not directly available due to abstraction. Here, the novel concept of state intersection plays an important role in maintaining a high level of information. Using this information, proving termination of complex algorithms is possible, as demonstrated in the annual termination competition. On grounds of the contributions presented in Chapter 1, AProVE was the most powerful tool in all competitions in the category for non-recursive JBC programs from 2009 to 2014. For a detailed evaluation and comparison with other tools, we again refer to [Bro14].

In Chapter 2, we discuss technical aspects of the technique presented in Chapter 1. As in the formalization many data structures of infinite size (most notably position sets for states containing cycles) are used, implementing algorithms using these data structures is far from trivial. In this chapter we present the algorithm used to merge states in the graph construction which only works on finite sets even if the position sets for the involved states are infinite.

In Chapter 3, we discuss how the approach of Chapter 1 can be extended to recursive programs. Here, the main problem is that when analyzing a recursive program, the call stack may grow without bounds. To solve this problem, we introduce a form of call stack abstraction where input arguments are used to replace abstracted parts of the heap. Using context concretization we then show how the analysis can be extended to also work on recursive states, so that the call stack height remains bounded. Here, as information corresponding to the call site is not explicitly represented in the states, side effects of the invoked method are propagated accordingly. Additionally, following the approach presented in Chapter 1, the technique is tuned to provide precise information in the resulting states even if the invoked method has side effects. Using these ideas, AProVE won the category for recursive JBC programs in all competitions from 2011 to 2014. In the competitions in 2009 and 2010 the ideas first presented in [BOG11] and extended in Chapter 3 were not implemented, yet.

In Chapter 4, we present a novel analysis making use of the information contained in Symbolic Execution Graphs. Here, we reason about the flow of information in the given program and, based on an initial marking provided by the user, identify parts of the program which could be left out without influencing the desired result. Based on the assumption that the user intended all parts of the program to be relevant for the computation, this analysis helps finding unintentional programming mistakes. As the Symbolic Execution Graph already contains information about sharing and aliasing effects on the heap, this analysis provides precise results without the need for additional complex computation.

In Chapters 1 and 3 parts of the presented techniques were published already. However, in this thesis we have adapted these techniques to a more general setting in which most of the concepts available in Java Bytecode may be represented (exceptions, static fields, etc.). Furthermore, the heap predicates used in this thesis are substantially more precise than the variants already presented. Most notably, the formalization of context concretization using heap predicates in this thesis is very involved, whereas in [BOG11] heap predicates were left out completely. In total, in this thesis the formalization of the techniques is much more complete, which also led to the discovery of several bugs.

Future Work

While the techniques presented in this thesis are designed to be as precise as possible, for termination analysis of real-world programs this is not always necessary. Instead, constructing Symbolic Execution Graphs for larger programs in a reasonable time is a huge challenge when trying to retain as much information as possible. In order to use this approach for a wider range of programs, some means of using a less precise abstraction must be found. In [CGJ$^+$00] the authors present an approach in which the level of abstraction is dynamically adapted based on where termination proofs fail and, thus, more information needs to be provided. Investigating if this approach can be adapted to the setting of this thesis seems to be promising.

In the case of recursive programs, the formalization of Chapter 3 should be completed. The open problem of input arguments abstraction needs to be solved in order to allow for a finite graph construction. Furthermore, the states once created for a specific recursive method may also be useful for a subsequent analysis of a different program calling the same method. By extending how the connection of call states and returning states as currently needed for context concretization is realized, the approach might become suitable for modularization.

Finally, the analysis presented in Chapter 4 should be formalized. An intuitive GUI should help users to actually find bugs. To show the power of the technique, one should also investigate a wide range of programs and try to find bugs.

A. Publications

[1] Carsten Fuhs, Rafael Navarro-Marset, Carsten Otto, Jürgen Giesl, Salvador Lucas, and Peter Schneider-Kamp. Search techniques for rational polynomial orders. In *Intelligent Computer Mathematics*, pages 109–124, Springer, 2008.

[2] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA '10)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 259–276. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010. Extended version (with proofs) appeared as technical report AIB-2010-08, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2010/2010-08.pdf`.

[3] Marc Brockschmidt, Carsten Otto, Christian von Essen, and Jürgen Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2010. Extended version (with proofs) appeared as technical report AIB-2010-15, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2010/2010-15.pdf`.

[4] Marc Brockschmidt, Carsten Otto, and Jürgen Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11)*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 155–170. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2011. Extended version (with proofs) appeared as technical report AIB-2011-02, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2011/2011-02.pdf`.

[5] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and NullPointerExceptions for Java Bytecode. In *Revised Selected Papers of the 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS '11)*, volume 7421 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 2012. Extended version (with proofs) appeared as technical report AIB-2011-19, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2011/2011-19.pdf`.

[6] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2012. Extended version (with proofs) appeared as technical report AIB-2012-06, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2012/2012-06.pdf`.

[7] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR '14)*, volume 8562 of *Lecture Notes in Artificial Intelligence*, pages 184–191. Springer, 2014.

B. Bibliography

[AAC+08] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination analysis of Java Bytecode. In *Proceedings of the 10th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2008.

[AG00] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[AHM+08] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, 2008.

[APV09] Saswat Anand, Corina S. Păsăreanu, and Willem Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1):53–67, 2009.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1985.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2007.

[Ben05] Nick Benton. Semantics of program analyses and transformations. *Lecture Notes for the PAT Summer School, Copenhagen*, 2005.

[BMOG12] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV '12)*, volume 7358 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2012. Extended version (with proofs) appeared as technical report AIB-2012-06, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2012/2012-06.pdf`.

[BN99] Franz Baader and Tobias Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1999.

[BOG11] Marc Brockschmidt, Carsten Otto, and Jürgen Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA '11)*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 155–170. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011. Extended version (with proofs) appeared as technical report AIB-2011-02, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2011/2011-02.pdf`.

[BOvEG10] Marc Brockschmidt, Carsten Otto, Christian von Essen, and Jürgen Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, volume 6463 of *Lecture Notes in Computer Science*, pages 17–37. Springer, 2010. Extended version (with proofs) appeared as technical report AIB-2010-15, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2010/2010-15.pdf`.

[Bro10] Marc Brockschmidt. The Finite Interpretation Graph: A versatile source for automated termination analysis of Java Bytecode. Diploma thesis. RWTH Aachen, 2010.

[Bro14] Marc Brockschmidt. Termination analysis for imperative programs operating on the heap. PhD thesis. RWTH Aachen, 2014.

[BSOG12] Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. Automated detection of non-termination and `NullPointerExceptions` for Java Bytecode. In *Revised Selected Papers of the 2nd International Conference on Formal Verification of Object-Oriented Software (FoVeOOS '11)*, volume 7421 of *Lecture Notes in Computer Science*, pages 123–141. Springer, 2012. Extended version (with proofs) appeared as technical report AIB-2011-19, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2011/2011-19.pdf`.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *ACM SIGPLAN Notices*, 44(1):289–300, 2009.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[CPR09] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.

[CRB10] Renato Cherini, Lucas Rearte, and Javier O. Blanco. A shape analysis for non-linear data structures. In *Proceedings of the 17th International Symposium on Static Analysis (SAS '10)*, volume 6337 of *Lecture Notes in Computer Science*, pages 201–217. Springer, 2010.

[FNMO+08] Carsten Fuhs, Rafael Navarro-Marset, Carsten Otto, Jürgen Giesl, Salvador Lucas, and Peter Schneider-Kamp. Search techniques for rational polynomial orders. In *Intelligent Computer Mathematics*, pages 109–124. Springer, 2008.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[Fro13] Florian Frohn. Modular termination analysis for Java Bytecode. Master thesis. RWTH Aachen, 2013.

[GBE+14] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR '14)*, volume 8562 of *Lecture Notes in Artificial Intelligence*, pages 184–191. Springer, 2014.

[GJS+12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. California, USA, 2012.

[GRS+11] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):7:1–7:39, 2011.

[GSS+12] Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. Symbolic evaluation graphs and term rewriting – a general methodology for analyzing logic programs. In *Proceedings of the 14th International Symposium on Principles and Practice of Declarative Programming (PPDP '12)*, pages 1–12. ACM Press, 2012.

[GSSKT06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. Automated termination analysis for haskell: From term rewriting to programming languages. In *Term Rewriting and Applications*, pages 297–312. Springer, 2006.

[Hic08] Rich Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on dynamic languages (DL '08)*. ACM Press, 2008.

[JLRS04] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. In *Static Analysis*, pages 246–264. Springer, 2004.

[KGK+07] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in action.* Manning Publications Co., 2007.

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19:385–394, 1976.

[KN06] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[Lan79] Dallas Lankford. *On Proving Term Rewriting Systems Are Noetherian.* Louisiana Tech Univ., Math. Department, 1979.

[Ler03] Xavier Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, 2003.

[LYBB12] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification.* Oracle America, Inc., 2012.

[MN70] Zohar Manna and Steven Ness. On the termination of markov algorithms. In *Proceedings of the Third Hawaii International Conference on System Science*, pages 789–792, 1970.

[NES$^+$11] Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 2011.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[OBvEG10] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA '10)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 259–276. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010. Extended version (with proofs) appeared as technical report AIB-2010-08, available online at `http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2010/2010-08.pdf`.

[ORY01] Peter O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL '01)*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[PR02] Samuele Pedroni and Noel Rappin. *Jython Essentials: Rapid Scripting in Java*. O'Reilly & Associates, Inc., 2002.

[RC11] Xavier Rival and Bor-Yuh Evan Chang. Calling context abstraction with shapes. *ACM SIGPLAN Notices*, 46(1):173–186, 2011.

[Rhi] `https://developer.mozilla.org/en-US/docs/Rhino`.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995.

[Sch08] Peter Schneider-Kamp. *Static Termination Analysis for Prolog using Term Rewriting and SAT Solving*. PhD thesis, RWTH Aachen, 2008.

[SG95] Morten H. Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In *Proceedings of ILPS'95, the International Logic Programming Symposium*. Citeseer, 1995.

[SGB$^+$14] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving termination and memory safety for programs with pointer arithmetic. In *Proceedings of the 7th International Joint Conference on Automated Reasoning (IJCAR '14)*, volume 8562 of *Lecture Notes in Artificial Intelligence*, pages 208–223. Springer, 2014.

[SGS$^+$10] Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, and René Thiemann. Automated termination analysis for logic programs with cut. *Theory and Practice of Logic Programming*, 10(4-6):365–381, 2010.

[SGST09] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated termination proofs for logic programs by term rewriting. *ACM Transactions on Computational Logic*, 11(1), 2009.

[SMP10] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyser for Java Bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3):8:1–8:70, 2010.

[TeR03] TeReSe. *Term Rewriting Systems*. Cambridge University Press, 2003.

[Thi07] René Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen, 2007.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2:230–265, 1936. Available online at `http://www.turingarchive.org/browse.php/B/12`.

[WZKSL13] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[YLB$^+$08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV '08)*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.

Curriculum Vitae

Name	Carsten Otto
Geburtsdatum	24. September 1983
Geburtsort	Tönisvorst

Bildungsgang

1994–2003	Bischöfliches Albertus-Magnus-Gymnasium Viersen-Dülken Abschluss: Allgemeine Hochschulreife
2003–2008	Studium der Informatik an der RWTH Aachen Abschluss: Diplom
2008–2013	Wissenschaftlicher Angestellter am Lehr- und Forschungsgebiet Informatik 2 (Prof. Dr. Jürgen Giesl), RWTH Aachen

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2012-01 Fachgruppe Informatik: Annual Report 2012

2012-02 Thomas Heer: Controlling Development Processes

2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

2012-04 Marcus Gelderie: Strategy Machines and their Complexity

2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting

2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

2012-07 André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms

2012-08 Hongfei Fu: Computing Game Metrics on Markov Decision Processes

2012-09 Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains

2012-10 Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations

2012-12 Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs

2012-15 Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations

2012-16 Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets

2012-17 Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods

2013-01 * Fachgruppe Informatik: Annual Report 2013

2013-02 Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen

2013-03 Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04 Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries

2013-05 Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06 Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation

2013-07 André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung

2013-08 Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers

2013-10 Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata

2013-12 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs

2013-13 Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

2013-14 Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures

2013-19 Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol

2013-20 Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models

2014-01 * Fachgruppe Informatik: Annual Report 2014

2014-02 Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software

2014-03 Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide

2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09	Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14	Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15	Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16	Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.