Automatic Generation of Specifications using Verification Tools

Automatische Spezifikationserzeugung mit Hilfe von Verifikationswerkzeugen
Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation
zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
vorgelegt von Dipl.-Inform. Nathan Daniel Wasser geboren in High Point, North Carolina, USA

Tag der Einreichung: 14.01.2016
Tag der Prüfung: 26.02.2016

1. Referent: Prof. Dr. Reiner Hähnle
2. Referent: Prof. Dr. Laura Kovács

Erscheinungsort: Darmstadt
Erscheinungsjahr: 2017

Darmstädter Dissertation — D 17

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Software Engineering

Automatic Generation of Specifications using Verification Tools
Automatische Spezifikationserzeugung mit Hilfe von Verifikationswerkzeugen

Zur Erlangung des Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Dissertation von Dipl.-Inform. Nathan Daniel Wasser aus High Point, North Carolina, USA

1. Referent: Prof. Dr. Reiner Hähnle
2. Referent: Prof. Dr. Laura Kovács

Tag der Einreichung: 14.01.2016
Tag der Prüfung: 26.02.2016

Erscheinungsort: Darmstadt
Erscheinungsjahr: 2017

Darmstädter Dissertation — D 17

Wissenschaftlicher Werdegang
Doktorand am Fachbereich Informatik der Technischen Universität Darmstadt
von Oktober 2009 bis September 2015

Studiengang Informatik: Diplom Informatiker (Dipl.-Inform.)
Technische Universität Darmstadt
von Oktober 2002 bis April 2009

Abstract

This dissertation deals with the automatic generation of sound specifications from a given program in the form of loop invariants and method contracts. Sound specifications are extremely useful, in that without them analysis of non-trivial programs becomes almost impossible. Verification tools can be used to prove complex properties for real-world programs, but this requires the presence of sound specifications for unbounded loops and unbounded recursive method calls. If even one simple specification is missing, the proof may become impossible to close.

In general automation and precision are two goals which are often mutually exclusive. To ensure that the generation of specifications is fully automatic, precision will suffer. Approaches exist which perform abstraction on programs, replacing all types with abstracted counterparts with only finitely many different abstract values. Thus algorithms relying on fixed points for these abstract values can be used in the automatic generation of specifications, ensuring termination thereof. Precision is lost not only at the loops and method calls where this is required to ensure automation, however, but in the entire program.

The automatic generation of specifications illustrated in this dissertation is characterized by the following: (i) abstraction is restricted to the loops and method calls themselves, ensuring that precision is kept for the remaining program, (ii) the loss of precision due to abstraction is partially reduced, by coupling the abstraction with introduction of new invariants which aim to counteract this loss of precision to a certain degree, and (iii) non-standard control flows of real-world programming languages are supported, rather than restricting the analysis to an academic toy language.

In order to restrict the loss of precision to loops and method calls, abstraction is performed on program states, rather than the entire program. This allows full precision to be kept where possible, while program states related to loops and method calls are abstracted in order to ensure the termination of fixed point algorithms. The abstraction of program states is performed using abstract domains for the corresponding types. These abstract values can then be used outside of the loop or method call as normal values for which only partial knowledge is present. Real-world programming languages, such as Java, can contain, for example, a program heap which can be modified in loops or method calls, as well as objects and arrays as types in addition to the simpler primitive types such as booleans and integers. This leads to abstract domains being presented for objects and program heaps.

As abstract domains are hard to fine-tune, additional invariants are introduced when abstracting, to counteract the coarse overapproximations. This allows abstraction of an array's elements, for example, by a coarse overapproximation of the program heap on which the elements reside, in addition to the introduction of invariants regarding the values of said array elements.

Real-world programming languages contain many elements that make the automatic generation of specifications much harder than these are on academic toy languages or strongly reduced subsets of real-world languages. Both loops and simple recursion are comparatively easy to reason about by themselves, however combining these, where a method calls itself recursively inside a loop, makes automatic generation of specifications a much harder task. Mutual recursion and non-standard control flows such as breaking out of a loop, throwing exceptions or returning from a method call while inside a loop add further complications. This dissertation describes how to automatically generate specifications in all of these cases.

Zusammenfassung

Diese Dissertation beschreibt die automatische Erzeugung korrekter Spezifikationen aus einem gegebenen Programm in Form von Schleifeninvarianten und Methodenverträge. Korrekte Spezifikationen sind für die Analyse nichttrivialer Programme unumgänglich. Verifikationswerkzeuge können komplexe Eigenschaften von realen Programmen beweisen, benötigen aber hierzu korrekte Spezifikationen für unbeschränkte Schleifen und unbeschränkte rekursive Methodenaufrufe. Selbst das Fehlen einer einzigen Spezifikation kann dazu führen, dass der Beweis nicht geschlossen werden kann.

Im Allgemeinen sind Automatisierung und Präzision Ziele, die sich oft gegenseitig ausschließen. Die Herausforderung besteht häufig darin einen guten Mittelweg zu finden. Um vollautomatische Erzeugung von Spezifikationen zu erlangen, müssen Kompromisse bei der Präzision hingenommen werden. Es existieren Ansätze, die Programme als Ganzes abstrahieren, indem alle Typen durch abstrakte Typen ersetzt werden, die nur endlich viele abstrakte Werte beinhalten. Dies ermöglicht es bei der automatischen Spezifikationserzeugung Algorithmen zu verwenden, die auf Fixpunktberechnung beruhen, und so deren Terminierung zu garantieren. Bei den bisherigen Ansätzen tritt ein Präzisionsverlust nicht nur bei Schleifen oder Methodenaufrufen auf, bei denen dieses zum Sicherstellen der Automatisierung notwendig ist, sondern im gesamten Programm.

Die in dieser Dissertation vorgestellte Spezifikationserzeugung zeichnet sich durch folgende Punkte aus: (i) Abstraktion findet nur bei Schleifen und Methodenaufrufen statt, so dass Präzision an allen anderen Stellen des Programms beibehalten wird, (ii) durch das Koppeln der Abstraktion an die Einführung neuer Invarianten wird eine präzisere Darstellung des symbolischen Programmzustandes erreicht, und (iii) werden komplexere Konstrukte zur Steuerung des Kontrollflusses realer Programmiersprachen unterstützt, anstatt sich auf einer akademischen Spielsprache zu beschränken.

Um den Präzisionsverlust zu minimieren, werden Programmzustände abstrahiert, an Stelle ganzer Programme. Um die Terminierung der Fixpunktalgorithmen sicherzustellen, müssen Programmzustände bei Schleifen oder Methodenaufrufen abstrahiert werden. Die Abstraktion von Programmzuständen wird mittels abstrakter Domänen passenden Typs ausgeführt. Diese abstrakten Werte können dann ausserhalb der Schleife oder des Methodenaufrufs als normale Werte betrachtet werden, für die nur Teilwissen zur Verfügung steht. Reale Programmiersprachen wie Java beinhalten zum Beispiel sowohl einen Programm-Heap, der von Schleifen oder Methodenaufrufen verändert werden kann, als auch Objekte und Arrays als Typen neben den Primitivtypen wie Bool'sche Werte oder ganze Zahlen. Abstrakte Domänen für Objekte und Programm-Heaps werden vorgestellt.

Da abstrakte Domänen sich nur schwer problemspezifisch anpassen lassen, werden zusätzliche Invarianten beim Abstrahieren eingeführt, um den Folgen der Überapproximation etwas entgegenzuarbeiten. Dies erlaubt es zum Beispiel Wissen über die Werte der Elemente eines Arrays beizubehalten, die ansonsten bei der Durchführung der Abstraktion verloren gegangen wären.

Reale Programmiersprachen beinhalten viele Konstrukte, die die automatische Spezifikationserzeugung erschweren. Sowohl bei Schleifen als auch bei einfacher Rekursion ist die automatische Spezifikationserzeugung vergleichsweise einfach zu realisieren. Kompliziertere Fälle, wie zum Beispiel wechselseitige Rekursion, stellen die automatische Spezifikationserzeugung vor signifikant größere Herausforderungen. Auch nichtstandard Kontrollflüsse, wie das Verlassen einer Schleife durch ein `break`, das Werfen einer Exception oder das Verlassen einer Methode innerhalb einer Schleife, sind weitere Komplikationen bei der automatischen Spezifikationserzeugung. Das in dieser Dissertation entwickelte Verfahren kann für alle diese Fälle korrekte Spezifikationen automatisch erzeugen.

Acknowledgements

This dissertation took time, effort, sweat, tears and me to the brink of madness. Having finished it is something I can forever look back on fondly and wear as a badge of pride. And it would *never* have been possible without the help, guidance, generosity and encouragement from the individuals I am about to name.

First and foremost I must thank my thesis advisor Reiner Hähnle, not merely because that is standard in such works, but for the far better reason that he is, without a doubt, *the* person without whom I would never have finished this dissertation. His stubbornness in believing in me surpassed even my determination to quit. Additionally, he was also a wonderful boss, great sounding board for bouncing ideas off of and an expert in a wide range of topics, including, but not limited to, good wine.

I thank Laura Kovács for taking the time to be my second reviewer and for the interesting conversation we had about some of the findings in this dissertation.

Before continuing with those most directly involved in supporting me in this writing business, overrated as it may be, a brief historical interlude: A long time ago, I began my university career under the guidance of Christoph Walther, thanks in large part to Markus Aderhold seeing in me a researcher, where I saw someone who never again wanted to see the inside of a university. In these first, formative years I learned quite a lot and met my first love, recursion. I also owe a great deal of thanks to Veronika Weber for helping me find my way during that time.

When Reiner took me into the SE group, I was overwhelmed by the friendly welcome. I immediately felt at home and will miss them all greatly. In particular, I would like to thank the core members of Tuesday night boardgaming, it was always a pleasure. I must also thank Martin Hentschel explicitly for making sure I never forgot to eat lunch. Gudrun Harris and the delicious goodies she brought in to work will be sorely missed. I also thank her for all the help and apologize if I ever stapled anything together. And I thank Richard Bubel for being a great researcher, person, dog watcher, colleague and friend. Talking over new ideas with Richard was one of the best ways to spend an afternoon and he always had time, even when he didn't. Aside from Reiner, Richard is most responsible for motivating me and helping me to bring out the best. This dissertation would not be half what it is without him. I wish him and the entire SE group all the best.

I thank the entire KeY community, in particular Daniel Grahl and Mattias Ulbrich, for the many interesting conversations, for helpful suggestions and for being a group of very nice people.

I would like to thank my parents, siblings and brother-in-law for their support during this stressful time, especially my sister Samantha Richtberg, for bugging me to finish and taunting me that she would finish her dissertation first. (Let's call it a draw.) I thank SharpMind, for putting my best interests ahead of theirs.

Finally, I thank my wonderful wife Marion and our two furry, four-legged children Coco and Dougie for being there for me when I needed them. Now is the time to repay that kindness with family time and dog treats.

Contents

List of Figures

List of Algorithms

List of Listings

1 Introduction

Correctness of software is an extremely important goal. Software bugs have been directly responsible for the deaths of over 100 people and billions of Euros in damages. With computers becoming more and more ubiquitous and software influencing our lives more than ever, the very real threats of faulty programs must be apparent.

As testing can prove only the presence of bugs and never their absence [19], formal specifications of program code and proofs that these formal specifications hold are one of the only guarantees for the correctness of software. But writing formal specifications is quite hard and time consuming for non-trivial programs. It requires many hours of a highly skilled professional's costly time. It also does not suffice to give a single specification of an entire program, as specifications are required for each unbounded loop and method call (or at least for the method's definition, akin to Bertrand Meyer's *design-by-contract* [48]). These specifications often allow verification tools to be used to prove complex properties for real-world programs automatically, or at least strongly reduce the amount of time a human needs to spend on the proof. But this requires that these specifications exist, which would normally call for a human to have invested time writing them. Here the idea of automatic specification generation comes in.

When it comes to the generation of specifications, automation and precision are two conflicting goals. If the generation of specifications should be fully automatic, precision will suffer. One such approach is *abstract interpretation* [15], which allows a fixed point algorithm for finding specifications, by abstracting all program types to abstract domains. Here precision is lost by the abstraction, while automation is gained.

Symbolic execution [45] is a technique where programs are executed not on concrete values, but rather symbolic values which may have constraints attached. This allows the execution of all possible inputs as a single symbolic execution run, which produces a symbolic exection tree. Approximation is required when encountering unbounded loops or method calls, by applying specifications for these. Outside of these approximations, symbolic execution is fully precise.

Combining symbolic execution with abstract interpretation allows the full precision of symbolic execution to be coupled with automatic abstraction of program states when unbounded loops and method calls are encountered. While the abstracted program states introduce a loss of precision, this need not be as large a loss, as in the remaining program the abstracted values can simply be treated as symbolic values and so full precision of the remaining program for the abstracted input is guaranteed.

This combination of symbolic execution and abstract interpretation was proposed in [12], where its use in automatically generating loop invariants was demonstrated for an academic toy language without method calls, nested loops or non-primitive types.

This dissertation describes the steps involved in applying this ingenious idea to a closer approximation of a real-world programming language. A symbolic execution engine for an almost complete subset of sequential Java has been described in [9], extended to use of an explicit heap in [63] and implemented in the state-of-the-art verification tool KeY[1] [1]. Applying the ideas from [12] to the scope of this language involves introducing abstract domains for all types which can be encountered, including objects, arrays and program heaps, as well as generating loop invariants for loops with non-standard control flows, such as breaking out of the loop, and generating method contracts for recursive method calls. Also required is the generation of specifications requiring further specifications, such as is the case with nested loops, mutual recursion and recursive calls from within a loop body.

As abstract domains are often too coarse, a finer grained solution is introduced by adding additional invariants when abstracting, to somewhat counteract the overapproximations. This allows abstraction of an array's elements, for example, by the combination of a coarse overapproximation of the program heap

[1] KeY can be downloaded at `http://www.key-project.org`

on which the elements reside with the introduction of invariants regarding the values of all elements within different partitions of the array.

The dissertation is structured in the following way:

- Chapter 2 gives background information on abstract interpretation, dynamic logic and the initial paper [12] that combined these ideas to automate loop invariant generation for an academic toy while-language.
- Chapter 3 gives abstract domains for the non-heap types which can appear in the programs we consider.
- Chapter 4 gives the abstract domain for program heaps we use, a widening operator for this abstract domain and a detailed proof for the correctness thereof.
- Chapter 5 explains how the coarse overapproximations resulting from abstraction can be fine-tuned with the addition of *invariant patterns* and describes these invariant patterns in detail. Thus, for example, abstraction of partitions of an array is possible.
- Chapter 6 explains how non-standard control flows within a loop body complicate not only the automatic generation of loop invariants, but also the application of a sound loop invariant rule. Problems with existing solutions are described and a novel solution introduced.
- Chapter 7 contains the algorithms for automatically generating specifications and describes how these work, in particular how complicating factors such as nested loops, mutual recursion and loops containing recursive calls are dealt with soundly.
- Chapter 8 contrasts the ideas in this dissertation with other approaches to automatic specification generation.
- Finally, Chapter 9 sums up the work's conclusions.

1.1 Publications this Thesis is Based on

- *A Theorem Prover Backed Approach to Array Abstraction.* Nathan Wasser and Richard Bubel. In *5th International Workshop on Invariant Generation.* Held as part of Vienna Summer of Logic, Vienna, Austria. 2014. [61]

  This paper introduces three new ideas: (i) It extends the value abstraction in [12] from primitive types to arrays, by giving an abstraction for all elements of the array within a given contiguous range. It also extends the information flow tracking ideas from [12], by (ii) adding arrays of dependencies for program variables of type array, storing for each element a set of program variables on whose initial values the value of the array element depends, and (iii) adding a *dependency stack* to store implicit dependencies caused by branching program instructions and have these implicit dependencies as well as the explicit dependencies be assigned to the matching dependency variable when a program variable is assigned a value.

  I was the main author of this paper. All three of the above mentioned ideas were mine.

- *Generating specifications for recursive methods by abstracting program states.* Nathan Wasser. In *Dependable Software Engineering: Theories, Tools, and Applications — First International Symposium, Nanjing, China.* 2015. [60]

  This paper uses an academic recursive toy language to introduce a way to automatically generate method contracts for recursive methods by repeated symbolic execution and abstraction of resulting pre- and postconditions for the recursive calls. An implementation of this approach using KeY is used to generate method contracts for selected recursive algorithms from the literature.

I was the sole author of this paper. The novel ideas put forth in the paper were all mine and evaluation of the implementation was performed by me.

- *Abstract Interpretation.* Nathan Wasser and Reiner Hähnle and Richard Bubel. In *Deductive Software Verification – The KeY Book: From Theory to Practice.* 2016. [62]

  In this chapter of the most recent version of the KeY book, value abstraction is described in the context of KeY. We mention useful abstract domains for primitives, and introduce novel abstract domains for objects and heaps.

  I came up with the novel abstract domains for objects and heaps, and made major contributions to this chapter.

- *Array Abstraction with Symbolic Pivots.* Reiner Hähnle and Nathan Wasser and Richard Bubel. In *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday.* 2016. [34]

  This paper introduces the novel idea of *symbolic pivots* to reason about array indices in loops. With the help of these, non-contiguous partitions of arrays can be abstracted. An integration of this approach in the implementation of loop invariant generation based on KeY allows for the generation of stronger loop invariants involving arrays.

  I was the main author of this paper. The idea behind symbolic pivots was mine. I performed the experiments.

1.2 Further Publications

- *Verification, Induction, Termination Analysis: Festschrift for Christoph Walther on the Occasion of His 60th Birthday.* 2010. *Simon Siegler and Nathan Wasser, editors.* [55]

  As one of the editors of the Festschrift for Christoph Walther, I was involved in gathering and reviewing paper submissions, working with the authors to fix any problems, and putting the Festschrift together.

- *Towards Fully Automatic Logic-Based Information Flow Analysis: An Electronic-Voting Case Study.* Quoc Huy Do and Eduard Kamburjan and Nathan Wasser. In *Principles of Security and Trust: 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings.* 2016. [22]

  This paper describes the combination of two logic-based tools in order to achieve almost full automation in analyzing information flow security while maintaining high precision. One of the tools is KeG [21], which allows Java programs annotated with information flow policies to be analyzed with respect to these policies. Information flow leaks detected are then exposed by a generated `JUnit` test. However, KeG requires loop invariants to be supplied for unbounded loops and method contracts to be supplied for unbounded recursive method calls. In order to provide these, an implementation based on the work in [61, 60, 34] automatically generates specifications for the input before passing both to KeG. An electronic voting case study utilizing this approach is discussed.

- *Fermat, Euler, Wilson - Three Case Studies in Number Theory.* Christoph Walther and Nathan Wasser. In *Journal of Automated Reasoning.* 2016. [59]

  This paper demonstrates how three well-known theorems from number theory can be proven using the verification tool VeriFun[2] [57, 58].

[2] VeriFun can be downloaded at `http://verifun.de`

1.3 Impact

This dissertation advances research in the fields of proof automation, specification generation, abstraction techniques for complex types such as program heaps and arrays, as well as dynamic logic [36].

The case study [22] demonstrates how the approach described in this dissertation can be incredibly useful, by automatically generating specifications required by, in this example, another verification tool dealing with information flow security. This validates the claim that advancements in automatic specification generation have wide reaching effects.

Further areas where the automatic generation of specifications described in this dissertation can be useful, are in (i) the integration with an IDE, such that the programmer's code is annotated with specifications on the fly, similar to the integration of automatic bug checking tools based on static analysis already present in state-of-the-art IDEs; and (ii) enhancing the work in [37], where a user supplied contract for a method implementation is attempted to be proven in a background task. If symbolic execution within this proof encounters an unbounded loop or method call without a user supplied specification, the proof cannot finish. By integrating the automatic generation of specifications in these cases, the proofs can be continued.

The work on generating specifications for non-standard control flows described in Chapter 6 revealed an inherent incompleteness in the dynamic logic calculus rule given in [9] for applying loop invariants on loops with non-standard control flows. The theory in [9] is further based on the introduction of a multitude of new modalities, which could be a nightmare to implement. Therefore the implementation in KeY was based on program transformation of the loop body, although program transformation of real-world programming languages is error-prone. Research on this topic resulted in the introduction of a new dynamic logic calculus rule to apply loop invariants. This solution requires neither the introduction of new modalities into dynamic logic, nor program transformation of the loop body, thus being both easier to implement and more likely to be implemented bug-free. The results have been shared with the KeY community, such that the newest release of the KeY book [2] can take these into account.

An implementation of some of the novel ideas in this dissertation was integrated into a branch of the KeY project. This branch was kept up-to-date with changes on the master branch, such that moving these new features to the master branch can be accomplished with minimal effort, once the feature freeze in preparation for the release of the new KeY book has ended.

2 Background

In this chapter we briefly introduce the concepts of *abstract interpretation* and *Java dynamic logic*, taking the information required in order to follow this dissertation from a few main sources while referring interested readers to further information sources if they are so inclined.

2.1 Basics

Definition 1 (Partial Order)**.** *Given a set M, a* partial order $\preceq$ *is a relation on M (i.e. a subset of $M \times M$) which is reflexive, anti-symmetric and transitive. I.e. for all $x, y, z \in M$:*

1. $x \preceq x$
2. $x = y$, *if* $x \preceq y$ *and* $y \preceq x$
3. $x \preceq z$, *if* $x \preceq y$ *and* $y \preceq z$

A set with a partial order is also called a *partially ordered set*, or *poset*.

Definition 2 (Join-semilattice)**.** *A set M with partial order $\preceq$ is a* join-semilattice *iff for all $x, y \in M$ there exists a least upper bound (or supremum) of the set $\{x, y\}$. We call the binary operator which calculates this least upper bound for x and y the* join operator*.*

Definition 3 (Sequence)**.** *Given a non-empty set M, an (infinite)* sequence $\langle x_i \rangle$ *is a function of type $\mathbb{N} \to M$. For all $n \in \mathbb{N}$ we write x_n to refer to $\langle x_i \rangle(n)$.*

Infinite sequences are often defined by recursion.

Definition 4 (Ultimately Stationary)**.** *A sequence $\langle x_i \rangle$ is* ultimately stationary *iff there exists some $n \in \mathbb{N}$, such that $x_n = x_m$ for all $m \in \mathbb{N}$, where $m > n$.*

Definition 5 (Ascending Chain)**.** *Given a set M with partial order $\preceq$, an* ascending chain *is a sequence $\langle x_i \rangle$, such that $x_n \preceq x_{n+1}$ for all $n \in \mathbb{N}$.*

Definition 6 (Infinite Ascending Chain)**.** *An* infinite ascending chain *is an ascending chain, which is not ultimately stationary.*

Definition 7 (Ascending Chain Condition)**.** *A partially ordered set satisfies the* ascending chain condition *iff all its ascending chains are ultimately stationary, i.e. it contains no infinite ascending chains.*

2.2 Abstract Interpretation

Abstract interpretation [15] is a formal verification technique which allows proving that the *abstract* semantics of a program satisfies an *abstract* specification. If the abstract semantics is sound, i.e. conclusions in the abstract semantics imply the same in the concrete semantics, then a proof via abstract interpretation guarantees that all concrete instances satisfy the abstract specification.

The heart of abstract interpretation is the concept of an *abstract domain*.

Definition 8 (Abstract Domain)**.** *Given a concrete domain D, an* abstract domain $\mathscr{A} = (A, \sqcup, \sqsubseteq)$ *is a join-semilattice, with join operator $\sqcup$ and partial order $\sqsubseteq$. There is an abstraction function $\alpha : 2^D \to A$ and a concretization function $\gamma : A \to 2^D$ which form a (monotone) Galois connection (initially introduced as antitone [10], the standard is now the monotone Galois connection [15]) , such that:*

1. $\forall X, Y \in 2^D.\ X \subseteq Y \to \alpha(X) \sqsubseteq \alpha(Y)$

2. $\forall a, b \in A.\ a \sqsubseteq b \to \gamma(a) \subseteq \gamma(b)$

3. $\forall X \in 2^D.\ X \subseteq \gamma(\alpha(X))$

4. $\forall a \in A.\ a = \alpha(\gamma(a))$

Let $f : A \to A$ be any function. The monotonic function $f' : A \to A$ is defined as $f'(a) = a \sqcup f(a)$. If $\mathscr{A}$ satisfies the ascending chain condition [6] (trivially the case if $\mathscr{A}$ has finite height), then starting with any initial input $x \in A$ a least fixed point [56] for f' on this input can be found by locating the stationary limit of the sequence $\langle x'_i \rangle$, where $x'_0 = x$ and $x'_{n+1} = f'(x'_n)$.

Abstract interpretation makes use of this when analyzing a program. Let `p` be a loop, `x` the only variable in `p` and $a \in A$ the abstract value of `x` before execution of the loop. Then we can see f as the abstract semantic function of a single loop iteration on the variable `x`. The fixed point for f' is an abstract value expressing an overapproximation of the set of all values of `x` before and after *each* iteration. Therefore it is sound to replace the loop with the assignment $\texttt{x} = a$.

If $\mathscr{A}$ does not satisfy the ascending chain condition, there may not be a stationary limit for $\langle x'_i \rangle$. In these cases a *widening operator* is required.

Definition 9 (Widening Operator $\cdot\nabla\cdot$)**.** *A* widening operator *for an abstract domain $\mathscr{A}$ is a function $\nabla : A \times A \to A$, where*

1. $\forall a, b \in A.\ a \sqsubseteq a \nabla b$

2. $\forall a, b \in A.\ b \sqsubseteq a \nabla b$

3. *for any sequence $\langle y'_n \rangle$ and initial value for x'_0 the sequence $\langle x'_n \rangle$ is ultimately stationary, where $x'_{n+1} = x'_n \nabla y'_n$.*

If $\mathscr{A}$ has a least element $\bot$, it suffices to use this as the initial value for x'_0, rather than proving the property for all possible initial values.

For further information on abstract interpretation, abstract domains and widening operators see [14, 15, 16].

2.3 Java Dynamic Logic

Java dynamic logic [9] is a sound extension of *dynamic logic* [36], which is a sound abstract semantics. The main idea of dynamic logic is to be able to express statements about program behavior by integrating deterministic programs and formulas within a single language. The *modalities* $\langle p \rangle$ and $[p]$ can be used in formulas, where p is any sequence of legal program statements. These operators refer to the final state of p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p]\phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For this dissertation we restrict ourselves to partial correctness and therefore mainly only use the box modality $[\cdot]$. For example, “when started in a state where `x` is zero, then if `x++` terminates it does so in a state where `x` is one” can be expressed in dynamic logic as $\texttt{x} \doteq 0 \to [\texttt{x++}](\texttt{x} \doteq 1)$. Java dynamic logic (Java DL) extends typed first-order logic. An extension of Java DL to explicit heap notation can be found in [63]. The following definitions regarding syntax and semantics are taken from [9, 63], modified and simplified slightly in order to give the reader enough basics to understand the following chapters.

Definition 10. *A* type hierarchy *is a pair* $\mathscr{T} = (\mathsf{TSym}, \sqsubseteq)$*, where*

1. TSym *is a set of type symbols, containing at least the empty type* $\bot$ *and the universal type* $\top$
2. $\sqsubseteq$ *is a reflexive, transitive relation on* TSym*, called the subtype relation;*
3. $\bot \sqsubseteq A \sqsubseteq \top$ *for all* $A \in \mathsf{TSym}$.

The type hierarchy $\mathscr{T}_J$ in Figure 2.1 is the minimal type hierarchy for Java DL. A Java DL type hierarchy for a given Java program *Prg* is any hierarchy $\mathscr{T} = (\mathsf{TSym}, \sqsubseteq)$ that contains $\mathscr{T}_J$ as a sub-hierarchy. All Java integer types (`byte`, `char`, `short`, `int`, `long`) are mapped to the type *int* $\in \mathsf{TSym}$. Floating-point numbers are not (yet) supported.

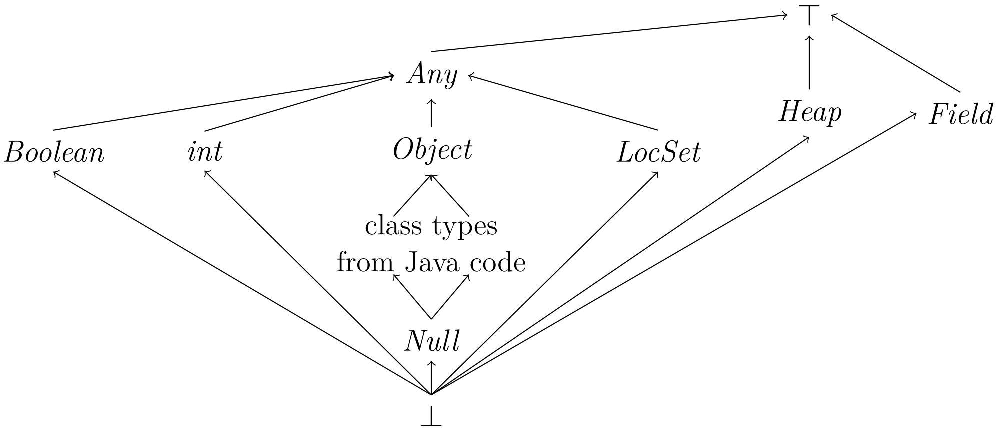


Figure 2.1: The minimal type hierarchy $\mathscr{T}_J$

Definition 11. *A type hierarchy* $\mathscr{T}_2 = (\mathsf{TSym}_2, \sqsubseteq_2)$ *is an* extension *of a type hierarchy* $\mathscr{T}_1 = (\mathsf{TSym}_1, \sqsubseteq_1)$*, in symbols* $\mathscr{T}_1 \sqsubseteq \mathscr{T}_2$*, if*

1. $\mathsf{TSym}_1 \subseteq \mathsf{TSym}_2$
2. $\sqsubseteq_2$ *is the smallest subtype relation containing* $\sqsubseteq_1 \cup \Delta$ *where* Δ *is a set of pairs* (S, T) *with* $T \in \mathsf{TSym}_1$ *and* $S \in \mathsf{TSym}_2 \setminus \mathsf{TSym}_1$.

Definition 12. *A* signature $\Sigma = (\mathsf{FSym}, \mathsf{PSym}, \mathsf{VSym})$ *for a given type hierarchy* $\mathscr{T}$ *is made up of the disjoint sets* $\mathsf{FSym}, \mathsf{PSym}, \mathsf{VSym}$*, where*

1. *the set* FSym *contains typed function symbols* $f : A_1, \ldots, A_n \to A$*,*
2. *the set* PSym *contains typed predicate symbols* $p(A_1, \ldots, A_n)$*, in particular at least the dedicated binary symbol* $\doteq(\top, \top)$ *for equality and the predicate symbols* $true()$ *and* $false()$*, and*
3. *the set* VSym *contains typed variable symbols* $\boldsymbol{v} : A$.

All types A*,* A_i *in this definition must be different from* $\bot$*. Function symbols without any arguments* $c : \ \to A$ *are called constant symbols of type* A*, while predicate symbols* $p()$ *are called propositional variables or propositional atoms. For sake of readability we can omit the empty parentheses.*

Java DL requires a minimum signature Σ_J given in [2]. An excerpt of this signature containing the elements required for this dissertation is shown in Figure 2.2, while some axioms for these are shown in Figure 2.3 and an excerpt of rules for *wellFormed* heaps is shown in Figure 2.4.

In Java DL, there are both *rigid* and *non-rigid* function symbols. While the interpretation of rigid symbols maintain their interpretation throughout program execution, non-rigid symbols can be changed by the program. In particular constant non-rigid symbols thus can be construed as *program variables.*

Definition 13. *Let $\mathscr{T}$ be a Java DL type hierarchy for a Java program Prg. A* Java DL signature *w.r.t. $\mathscr{T}$ is a tuple*

$$\Sigma = (\mathsf{FSym}_r, \mathsf{FSym}_{nr}, \mathsf{PSym}, \mathsf{VSym})$$

where

- FSym_r *and* FSym_{nr} *are disjoint sets of function symbols;*
- $(\mathsf{FSym}_r, \mathsf{PSym}, \mathsf{VSym})$ *includes at least the vocabulary from* Σ_J*;*
- *the set* $\mathsf{PVSym} \subseteq \mathsf{FSym}_{nr}$ *of all constant non-rigid function symbols, which we call program variables, contains all local variables* `a` *declared in Prg, where the type of* $\mathtt{a} : A \in \mathsf{PVSym}$ *is given by the declared Java type T as follows:*
  - $A = T$ *if T is a reference type,*
  - $A = Boolean$ *if* $T = \mathtt{boolean}$,
  - $A = int$ *if* $T \in \{\mathtt{byte}, \mathtt{char}, \mathtt{short}, \mathtt{int}, \mathtt{long}\}$.
- $\mathsf{PVSym} \subseteq \mathsf{FSym}_{nr}$ *contains an infinite number of symbols of every typing.*
- $\mathsf{PVSym} \subseteq \mathsf{FSym}_{nr}$ *contains the "special" program variable* $\mathtt{heap} : Heap \in \mathsf{PVSym}$.

The set of all (rigid and the non-rigid) function symbols is denoted by $\mathsf{FSym} = \mathsf{FSym}_r \cup \mathsf{FSym}_{nr}$.

While *logical variables* in VSym can be universally or existentially quantified but never occur in programs, *program variables* in PVSym can occur in programs but cannot be quantified.

Definition 14 (Method Identifier)**.** *For any Java DL type hierarchy* $(\mathsf{TSym}, \sqsubseteq)$ *and Java program Prg, let T be the set of types, such that*

$$\text{for all } \tau \text{ in } \mathsf{TSym},\ \tau \text{ is in } T \text{ iff } \tau \text{ is } int \text{ or } \tau \text{ is } Boolean \text{ or } Null \sqsubset \tau \sqsubseteq Object$$

Then a method identifier *is a tuple* $(\mathtt{m}, sig, t, C)$*, where* `m` *is the method name,* $sig \in (T \times \cdots \times T)$ *is the method signature,* $t \in (T \cup \{\mathtt{void}\})$ *is the method's return type and C is the Java class in Prg containing an implementation of the method. We write the method identifier* $(\mathtt{m}, sig, t, C)$ *as*

$$\mathtt{m}(sig \to t)\mathtt{@}C$$

Definition 15 (Legal program fragments)**.** *Let Prg be a Java program. A* legal program fragment *contained in a modality in Java DL is a sequence of Java statements, where there are local variables* $\mathtt{a}_1, \ldots, \mathtt{a}_n \in \mathsf{PVSym}$ *of Java types* $T_1, \ldots, T_n$ *such that extending Prg with an additional class*

```
class C {
    static void m(T_1 a_1, ..., T_n a_n) { p }
}
```

yields again a legal program according to the rules of the Java language specification, except that

- *p may refer to fields, methods and classes that are not visible in* `C`*, and*

int and *Boolean*	all function and predicate symbols for *int*, e.g., $+, *, <, \ldots$
	Boolean constants TRUE, FALSE
Java types	$\mathit{null} : \mathit{Null}$
	$\mathit{length} : \mathit{Object} \to \mathit{int}$
	$\mathit{cast}_A : \mathit{Object} \to A$ for any A in $\mathcal{T}$ with $\bot \sqsubset A \sqsubseteq \mathit{Object}$.
	$\mathit{instance}_A : \mathit{Any} \to \mathit{Boolean}$ for any type $A \sqsubseteq \mathit{Any}$
	$\mathit{exactInstance}_A : \mathit{Any} \to \mathit{Boolean}$ for any type $A \sqsubseteq \mathit{Any}$
Field	$\mathit{created} : \mathit{Field}$
	$\mathit{arr} : \mathit{int} \to \mathit{Field}$
	$f : \mathit{Field}$ for every Java field f
Heap	$\mathit{select}_A : \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \to A$ for any type $A \sqsubseteq \mathit{Any}$
	$\mathit{store} : \mathit{Heap} \times \mathit{Object} \times \mathit{Field} \times \mathit{Any} \to \mathit{Heap}$
	$\mathit{create} : \mathit{Heap} \times \mathit{Object} \to \mathit{Heap}$
	$\mathit{wellFormed}(\mathit{Heap})$
LocSet	$\dot{\in}(\mathit{Object}, \mathit{Field}, \mathit{LocSet})$
	$\dot{\emptyset} : \mathit{LocSet}$
	$\mathit{allLocs} : \mathit{LocSet}$
	$\mathit{singleton} : \mathit{Object} \times \mathit{Field} \to \mathit{LocSet}$
	$\dot{\cup} : \mathit{LocSet} \times \mathit{LocSet} \to \mathit{LocSet}$
	$\mathit{allFields} : \mathit{Object} \to \mathit{LocSet}$
	$\mathit{allObjects} : \mathit{Field} \to \mathit{LocSet}$
	$\mathit{unusedLocs} : \mathit{Heap} \to \mathit{LocSet}$
	$\mathit{anon} : \mathit{Heap} \times \mathit{LocSet} \times \mathit{Heap} \to \mathit{Heap}$

Figure 2.2: An excerpt of the vocabulary Σ_J

$$\forall \mathit{Object}\ x; (\mathit{instance}_A(x) \doteq \mathit{TRUE} \leftrightarrow \exists y : A; (y \doteq x))$$
$$\forall \mathit{Object}\ x; (\mathit{exactInstance}_A(x) \doteq \mathit{TRUE} \to \mathit{instance}_A(x) \doteq \mathit{TRUE})$$
$$\forall \mathit{Object}\ x; (\mathit{exactInstance}_A(x) \doteq \mathit{TRUE} \to \mathit{instance}_B(x) \doteq \mathit{FALSE}) \text{ with } A \not\sqsubseteq B$$
$$\forall \mathit{Object}\ x; (\mathit{instance}_A(x) \doteq \mathit{TRUE} \to \mathit{cast}_A(x) \doteq x)$$
$$\forall \mathit{Object}\ x; (\mathit{length}(x) \geq 0)$$

Figure 2.3: Axioms for functions related to Java types

onlyCreatedObjectsAreReferenced
$wellFormed(h) \rightarrow select_A(h,o,f) \doteq null \vee select_{boolean}(h, select_A(h,o,f), created) \doteq TRUE$

wellFormedStoreObject
$$wellFormed(h) \wedge (x \doteq null \vee (select_{boolean}(h,x,created) \doteq TRUE \wedge instance_A(x) \doteq TRUE)) \\ \rightarrow wellFormed(store(h,o,f,x)) \quad \text{where } f \text{ is declared as a field of type} A$$

wellFormedStorePrimitive
$wellFormed(h) \rightarrow wellFormed(store(h,o,f,x))$
provided f is a field of type A, x is of type $B,$ and $B \sqsubseteq A, B \not\sqsubseteq Object, B \not\sqsubseteq LocSet$

wellFormedStorePrimitiveArray
$wellFormed(h) \rightarrow wellFormed(store(h,o,arr(idx),x))$
provided o is of sort A, x is of sort $B, B \not\sqsubseteq Object, B \not\sqsubseteq LocSet, B \sqsubseteq A$

wellFormedCreate
$wellFormed(h) \rightarrow wellFormed(create(h,o))$

wellFormedAnon
$wellFormed(h) \wedge wellFormed(h2) \rightarrow wellFormed(anon(h,y,h2))$

The variables $h, h_2 : Heap$, $o : Object$, $f : Field$, $x : Any$, $y : LocSet$ are implicitly universally quantified.

Figure 2.4: An excerpt of rules for the predicate *wellFormed*

- *p may contain method frames in addition to normal Java statements. A method frame is a statement of the form*

  `method-frame(source=`m`, result->`r`, this=`t`) : {` *body* `} ,`

  where (a) m is a method identifier, (b) r is a local variable, (c) t is an expression free from side-effects and from method calls, and (d) body is a legal program fragment in the context of Prg. The semantics of a method frame is that, inside body (but outside of any nested method frames that might be contained in body), the keyword `this` *evaluates to the value of t, and the intent of a* `return` *statement is to assign the returned value to r and to then exit the method frame. Both "*`result->`r*" and "*`this=`t*" are optional. A method-frame without a reference to* `this` *denotes a* `static` *method, while a missing* `result` *pointer indicates either that the method is declared* `void` *and therefore has no return value, or that the process of returning is already underway.*

In addition to *legal program fragments*, Java DL contains *terms*, *formulas* and *updates*. As these are mutually defined and the notion of terms and formulas is clear, we first present the idea and definition of updates and then give the formal definition for terms and formulas. Like program fragments, updates denote state changes. The difference between updates and program fragments is that updates are a simpler and more restricted concept. For example, updates always terminate, and the expressions occurring in updates never have side effects.

Definition 16 (Updates)**.** *Let Prg be a Java program, $\mathcal{T}$ a type hierarchy for Prg, and Σ a signature for $\mathcal{T}$. The set* Upd *of updates is inductively defined*

- $(\mathtt{a} := t) \in \mathsf{Upd}$ *for each program variable* $\mathtt{a} : A \in \mathsf{PVSym}$ *and term* $t \in \mathsf{Trm}_{A'}$ *such that* $A' \sqsubseteq A$.
- $(\mathcal{U}_1 \,||\, \mathcal{U}_2) \in \mathsf{Upd}$ *for all updates* $\mathcal{U}_1, \mathcal{U}_2 \in \mathsf{Upd}$.
- $(\{\mathcal{U}_1\}\mathcal{U}_2) \in \mathsf{Upd}$ *for all updates* $\mathcal{U}_1, \mathcal{U}_2 \in \mathsf{Upd}$.

Intuitively, an *elementary update* $\mathtt{a} := t$ assigns the value of the term t to the program variable $\mathtt{a}$, a *parallel update* $\mathcal{U}_1 \,||\, \mathcal{U}_2$ performs assignment of the updates $\mathcal{U}_1$ and $\mathcal{U}_2$ in parallel (where the elementary update in $\mathcal{U}_2$ wins in case of clashes), while an *update application* $\{\mathcal{U}\}\xi$ (where ξ can be a term, formula, or update) has the inherent meaning that ξ should be evaluated in the state produced by $\mathcal{U}$.

Definition 17 (Terms and Formulas of Java DL)**.** *Let Prg be a Java program,* $\mathcal{T}$ *a type hierarchy for Prg, and* Σ *a signature w.r.t.* $\mathcal{T}$*.*

The set Trm_A *of Java DL terms of type* A*, for* $A \neq \bot$*, is inductively defined by:*

1. $v \in \mathsf{Trm}_A$ *for each variable symbol* $v : A \in \mathsf{VSym}$ *of type* A*.*
2. $f(t_1, \ldots, t_n) \in \mathsf{Trm}_A$ *for each* $f : A_1, \ldots, A_n \to A \in \mathsf{FSym}$ *and all terms* $t_i \in \mathsf{Trm}_{B_i}$ *with* $B_i \sqsubseteq A_i$ *for* $1 \leq i \leq n$*.*
3. $(\text{if } \phi \text{ then } t_1 \text{ else } t_2) \in \mathsf{Trm}_A$ *for* $\phi \in \mathsf{Fml}$ *and* $t_i \in \mathsf{Trm}_{A_i}$ *where* $A_2 \sqsubseteq A_1 = A$ *or* $A_1 \sqsubseteq A_2 = A$*.*
4. $\{\mathcal{U}\}t \in \mathsf{Trm}_A$ *for all updates* $\mathcal{U} \in \mathsf{Upd}$ *and all terms* $t \in \mathsf{Trm}_A$*.*

The set Fml *of Java DL formulas is inductively defined by:*

1. $p(t_1, \ldots, t_n) \in \mathsf{Fml}$ *for* $p(A_1, \ldots, A_n) \in \mathsf{PSym}$*, and* $t_i \in \mathsf{Trm}_{B_i}$ *with* $B_i \sqsubseteq A_i$ *for all* $1 \leq i \leq n$*.*
2. $(\neg\phi)$*,* $(\phi \wedge \psi)$*,* $(\phi \vee \psi)$*,* $(\phi \to \psi)$*,* $(\phi \leftrightarrow \psi)$ *are in* Fml *for arbitrary* $\phi, \psi \in \mathsf{Fml}$*.*
3. $\forall v; \phi$*,* $\exists v; \phi$ *are in* Fml *for* $\phi \in \mathsf{Fml}$ *and* $v : A \in \mathsf{VSym}$*.*
4. $\langle p \rangle \phi$*,* $[p]\phi \in \mathsf{Fml}$ *for all legal program fragments* p*.*
5. $\{\mathcal{U}\}\phi \in \mathsf{Fml}$ *for all formulas* $\phi \in \mathsf{Fml}$ *and updates* $\mathcal{U} \in \mathsf{Upd}$*.*

A term or formula is called rigid if it does not contain any non-rigid function symbols.

2.3.2 Semantics of Java Dynamic Logic

Definition 18 (Domain)**.** *The* domain *for a given type hierarchy* $\mathcal{T}$ *and signature* Σ *consists of*

1. *a set* D*,*
2. *a typing function* $\delta : D \to \mathsf{TSym} \setminus \{\bot\}$ *such that for every* $A \in \mathsf{TSym}$ *the following set is not empty:*

$$D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$$

The set $D^A = \{d \in D \mid \delta(d) \sqsubseteq A\}$ *is called the type domain for* A*. This implies that for different types* $A, B \in \mathsf{TSym} \setminus \{\bot\}$ *there is an element* $o \in D^A \cap D^B$ *only if there exists* $C \in \mathsf{TSym}$*,* $C \neq \bot$ *with* $C \sqsubseteq A$ *and* $C \sqsubseteq B$*.*

The fixed type domains for Java DL are shown in Figure 2.5

Definition 19. *A* first-order structure $\mathcal{M}$ *for a given type hierarchy* $\mathcal{T}$ *and signature* Σ *consists of*

- *a domain* (D, δ)*,*
- *an interpretation* I

such that

1. $I(f)$ *is a function from* $D^{A_1} \times \cdots \times D^{A_n}$ *into* D^A *for* $f : A_1, \ldots, A_n \to A$ *in* FSym*,*

2. *$I(p)$ is a subset of $D^{A_1} \times \cdots \times D^{A_n}$ for $p(A_1, \ldots, A_n)$ in* PSym,

3. *$I(\doteq) = \{(d, d) \mid d \in D\}$.*

Some semantics are fixed for all first-order structures considered in Java DL. Fixed semantics are given in Figure 2.6 for the subset of Σ_J shown in Figure 2.2. Some important notes: The semantics of the *store* function is such that it cannot change the implicit field *created*. The function $anon^{\mathscr{M}}(h_1, s, h_2)$ overwrites the function h_1 for the arguments $(o, f) \in s$ by the values of h_2. This cannot lead to a *created* field being reset, but can set the *created* field of an unused location in h_1. Usually the *anon* operation is applied with h_2 being a heap function of unknown properties, such that h_1 is *anonymized* for the locations in s. The *length* function is defined for all elements in D^{Object}, not only for elements in D^{OT} where OT is an array type; however, only for arrays does the value of *length* have any greater meaning.

Definition 20. *Let $\mathscr{M}$ be a first-order structure with domain D.*

A variable assignment is a function $\beta : \mathsf{VSym} \to D$ such that $\beta(v) \in D^A$ for $v : A \in \mathsf{VSym}$.

For a variable assignment β, a variable $v : A \in \mathsf{VSym}$ and a domain element $d \in D^A$, a modified variable assignment is defined as:

$$\beta_v^d(v') = \begin{cases} d & \text{if } v' = v \\ \beta(v') & \text{if } v' \neq v \end{cases}$$

A term t can be evaluated with regard to a first-order structure $\mathscr{M}$ and variable assignment β. The notation for this is $\mathsf{val}_{\mathscr{M},\beta}(t)$. A formula ϕ can be defined to be true (or false) with respect to $\mathscr{M}$ and β The notation for this is $(\mathscr{M}, \beta) \models \phi$ (or $(\mathscr{M}, \beta) \not\models \phi$).

Definition 21. *The evaluation of terms $t \in \mathsf{Trm}_A$ is defined inductively by:*

- *$\mathsf{val}_{\mathscr{M},\beta}(v) = \beta(v)$ for any variable v.*
- *$\mathsf{val}_{\mathscr{M},\beta}(f(t_1, \ldots, t_n)) = I(f)(\mathsf{val}_{\mathscr{M},\beta}(t_1), \ldots, \mathsf{val}_{\mathscr{M},\beta}(t_n))$.*
- *$\mathsf{val}_{\mathscr{M},\beta}(\text{if } \phi \text{ then } t_1 \text{ else } t_2) = \begin{cases} \mathsf{val}_{\mathscr{M},\beta}(t_1) & \text{if } (\mathscr{M}, \beta) \models \phi \\ \mathsf{val}_{\mathscr{M},\beta}(t_2) & \text{if } (\mathscr{M}, \beta) \not\models \phi \end{cases}$*

1. $D^{int} = \mathbb{Z}$,
2. $D^{Boolean} = \{tt, ff\}$,
3. $D^{ObjectType}$ is an infinite set of elements for every $ObjectType$ with $Null \sqsubset ObjectType \sqsubseteq Object$,
4. $D^{Null} = \{null\}$,
5. D^{Heap} = the set of all functions $h : D^{Object} \times D^{Field} \to D^{Any}$,
6. D^{LocSet} = the powerset of $\{(o, f) \mid o \in D^{Object}$ and $f \in D^{Field}\}$,
7. D^{Field} is an infinite set.

Figure 2.5: Semantics of fixed type domains

1. $\mathit{TRUE}^{\mathcal{M}} = \mathit{true}$ and $\mathit{FALSE}^{\mathcal{M}} = \mathit{false}$

2. $\mathit{select}_A^{\mathcal{M}}(h, o, f) = \mathit{cast}_A^{\mathcal{M}}(h(o, f)), \qquad \mathit{cast}_A^{\mathcal{M}}(o) = \begin{cases} o & \text{if } o \in A^{\mathcal{M}} \\ \text{arbitrary element in } A^{\mathcal{M}} & \text{otherwise} \end{cases}$

3. $\mathit{store}^{\mathcal{M}}(h, o, f, x) = h^*$, where the function h^* is defined by
$$h^*(o', f') = \begin{cases} x & \text{if } o' = o, f = f' \text{ and } f \neq \mathit{created}^{\mathcal{M}} \\ h(o', f') & \text{otherwise} \end{cases}$$

4. $\mathit{create}^{\mathcal{M}}(h, o) = h^*$, where the function h^* is defined by
$$h^*(o', f) = \begin{cases} \mathit{true} & \text{if } o' = o, o \neq \mathit{null} \text{ and } f = \mathit{created}^{\mathcal{M}} \\ h(o', f) & \text{otherwise} \end{cases}$$

5. $h \in \mathit{wellFormed}^{\mathcal{M}}$ iff (a) if $h(o, f) \in D^{\mathit{Object}}$, then $h(o, f) = \mathit{null}$ or $h(h(o, f), \mathit{created}^{\mathcal{M}}) = \mathit{tt}$, (b) if $h(o, f) \in D^{\mathit{LocSet}}$, then $h(o, f) \cap \mathit{unusedLocs}^{\mathcal{M}}(h) = \emptyset$, (c) there are only finitely many $o \in D^{\mathit{Object}}$ for which $h(o, \mathit{created}^{\mathcal{M}}) = \mathit{tt}$

6. $\mathit{arr}^{\mathcal{M}}$ is an injective function from $\mathbb{Z}$ into $\mathit{Field}^{\mathcal{M}}$

7. $\mathit{created}^{\mathcal{M}}$ and $f^{\mathcal{M}}$ for each Java field f are elements of $\mathit{Field}^{\mathcal{M}}$, which are pairwise different and also not in the range of $\mathit{arr}^{\mathcal{M}}$.

8. $\mathit{null}^{\mathcal{M}} = \mathit{null}$

9. $\mathit{instance}_A^{\mathcal{M}} = A^{\mathcal{M}} = \{o \in M \mid \delta(o) \sqsubseteq A\}$

10. $\mathit{exactInstance}_A^{\mathcal{M}} = \{o \in M \mid \delta(o) = A\}$

11. $\mathit{length}^{\mathcal{M}}(o) \in \mathbb{N}$

12. $\dot{\in}^{\mathcal{M}}(o, f, s)$ iff $(o, f) \in s$

13. $\dot{\emptyset}^{\mathcal{M}} = \emptyset$

14. $\mathit{allLocs}^{\mathcal{M}} = \mathit{Object}^{\mathcal{M}} \times \mathit{Field}^{\mathcal{M}}$

15. $\mathit{singleton}^{\mathcal{M}}(o, f) = \{(o, f)\}$

16. $\dot{\cup}^{\mathcal{M}}(s_1, s_2) = s_1 \cup s_2$

17. $\mathit{allFields}^{\mathcal{M}}(o) = \{(o, f) \mid f \in \mathit{Field}^{\mathcal{M}}\}$

18. $\mathit{allObjects}^{\mathcal{M}}(f) = \{(o, f) \mid o \in \mathit{Object}^{\mathcal{M}}\}$

19. $\mathit{unusedLocs}^{\mathcal{M}}(h) = \{(o, f) \mid o \in \mathit{Object}^{\mathcal{M}}, f \in \mathit{Field}^{\mathcal{M}}, o \neq \mathit{null}, h(o, \mathit{created}^{\mathcal{M}}) = \mathit{false}\}$

20. $\mathit{anon}^{\mathcal{M}}(h_1, s, h_2) = h^*$, where the function h^* is defined by:
$$h^*(o, f) = \begin{cases} h_2(o, f) & \text{if } (o, f) \in s \text{ and } f \neq \mathit{created}^{\mathcal{M}}, \text{ or} \\ & \qquad (o, f) \in \mathit{unusedLocs}^{\mathcal{M}}(h_1) \\ h_1(o, f) & \text{otherwise} \end{cases}$$

Figure 2.6: Semantics for the vocabulary from Figure 2.2

Definition 22. *The evaluation of formulas $\phi \in \mathsf{Fml}$ is inductively defined by:*

$$\begin{array}{lll}
1 & (\mathscr{M}, \beta) \models \mathit{true}, \ \ (\mathscr{M}, \beta) \not\models \mathit{false} & \\
2 & (\mathscr{M}, \beta) \models p(t_1, \ldots, t_n) & \textit{iff} \ \ (\mathsf{val}_{\mathscr{M},\beta}(t_1), \ldots, \mathsf{val}_{\mathscr{M},\beta}(t_n)) \in I(p) \\
3 & (\mathscr{M}, \beta) \models \neg\phi & \textit{iff} \ \ (\mathscr{M}, \beta) \not\models \phi \\
4 & (\mathscr{M}, \beta) \models \phi_1 \wedge \phi_2 & \textit{iff} \ \ (\mathscr{M}, \beta) \models \phi_1 \ \textit{and} \ (\mathscr{M}, \beta) \models \phi_2 \\
5 & (\mathscr{M}, \beta) \models \phi_1 \vee \phi_2 & \textit{iff} \ \ (\mathscr{M}, \beta) \models \phi_1 \ \textit{or} \ (\mathscr{M}, \beta) \models \phi_2 \\
6 & (\mathscr{M}, \beta) \models \phi_1 \rightarrow \phi_2 & \textit{iff} \ \ (\mathscr{M}, \beta) \not\models \phi_1 \ \textit{or} \ (\mathscr{M}, \beta) \models \phi_2 \\
7 & (\mathscr{M}, \beta) \models \phi_1 \leftrightarrow \phi_2 & \textit{iff} \ \ ((\mathscr{M}, \beta) \models \phi_1 \ \textit{and} \ (\mathscr{M}, \beta) \models \phi_2) \ \textit{or} \\
 & & \phantom{\textit{iff}} \ \ ((\mathscr{M}, \beta) \not\models \phi_1 \ \textit{and} \ (\mathscr{M}, \beta) \not\models \phi_2) \\
8 & (\mathscr{M}, \beta) \models \forall\ A\ v; \phi & \textit{iff} \ \ (\mathscr{M}, \beta_v^d) \models \phi \ \textit{for all} \ d \in D^A \\
9 & (\mathscr{M}, \beta) \models \exists\ A\ v; \phi & \textit{iff} \ \ (\mathscr{M}, \beta_v^d) \models \phi \ \textit{for at least one} \ d \in D^A
\end{array}$$

For a set of formulas Φ, $(\mathscr{M}, \beta) \models \Phi$ is short for: $(\mathscr{M}, \beta) \models \phi$ for all $\phi \in \Phi$. If ϕ contains no free variables, $(\mathscr{M}, \beta) \models \phi$ is the same for all variable assignment β and so the notation $\mathscr{M} \models \phi$ is used instead.

As Java allows inheritance, in order to make a claim about Java programs without restricting ourselves to a closed-world assumption, *logical consequence* requires that a formula hold in all type hierarchy extensions.

Definition 23. *Let $\mathscr{T}$ be a type hierarchy and Σ a signature, $\phi \in \mathsf{Fml}_{\mathscr{T},\Sigma}$ a formula without free variables, and $\Phi \subseteq \mathsf{Fml}_{\mathscr{T},\Sigma}$ a set of formulas without free variables.*

1. *ϕ is a logical consequence of Φ, in symbols $\Phi \models \phi$ if, for all type hierarchies $\mathscr{T}'$ with $\mathscr{T} \sqsubseteq \mathscr{T}'$ and all $\mathscr{T}'$-Σ-structures $\mathscr{M}$ such that $\mathscr{M} \models \Phi$, also $\mathscr{M} \models \phi$ holds.*

2. *ϕ is universally valid if it is a logical consequence of the empty set, i.e., if $\emptyset \models \phi$.*

3. *ϕ is satisfiable if there is a type hierarchy $\mathscr{T}'$, with $\mathscr{T} \sqsubseteq \mathscr{T}'$ and a $\mathscr{T}'$-Σ-structure $\mathscr{M}$ with $\mathscr{M} \models \phi$.*

Java DL formulas are evaluated in Kripke structures, which are collections of first-order structures. We take the following explanations and definitions from [2, Chapter 3]:

> *‘ Different first-order structures within a Kripke structure assign different values to program variables. Accordingly, they are called program states or simply states. We demand that states in the same Kripke structure differ only in the interpretation of the non-rigid function symbols (i.e., program variables). Two different Kripke structures, on the other hand, may differ in the choice of domain or interpretation of the predicate and rigid function symbols.*
>
> **Definition 24** (Java DL Kripke structure [2, Chapter 3])**.** *Let Prg be a Java program, $\mathscr{T}$ a type hierarchy for Prg and Σ a signature w.r.t. $\mathscr{T}$. A* Java DL Kripke structure *for Σ is a tuple*
>
> $$\mathscr{K} = (\mathscr{S}, \varrho)$$
>
> *consisting of*
>
> - *an infinite set $\mathscr{S}$ of first-order structures over Σ (Def. 19), which we will call states, such that:*
>   - *Any two states $s_1, s_2 \in \mathscr{S}$ coincide in their domain and in the interpretation of predicate and rigid function symbols.*
>   - *$\mathscr{S}$ is closed under the above property, i.e., any FOL structure coinciding with the states in $\mathscr{S}$ in the domain and the interpretation of the non-rigid function symbols is also in $\mathscr{S}$.*

- *a function ϱ that associates with every legal program fragment p a transition relation $\varrho(p) \subseteq \mathscr{S}^2$ such that $(s_1, s_2) \in \varrho(p)$ iff p, when started in s_1, terminates normally in s_2 (i.e., not by throwing an exception). (We consider Java programs to be deterministic, so for all legal program fragments p and all $s_1 \in \mathscr{S}$, there is at most one s_2 such that $(s_1, s_2) \in \varrho(p)$.)*

Definition 25 (Semantics of Java DL updates [2, Chapter 3])**.** *Let Prg be a Java program, $\mathscr{T}$ a type hierarchy for Prg, Σ a signature for $\mathscr{T}$, $\mathscr{K}$ a Kripke structure for Σ, $s \in \mathscr{S}$ a state, and $\beta : \mathsf{VSym} \to D$ a variable assignment. The valuation function $val_{\mathscr{K},s,\beta} : \mathsf{Upd} \to (\mathscr{S} \to \mathscr{S})$ is defined as follows:*

$$val_{\mathscr{K},s,\beta}(\{u\}t) = val_{\mathscr{K},s',\beta}(t),\ \textit{where}\ s' = val_{\mathscr{K},s,\beta}(u)(s)$$

$$val_{\mathscr{K},s,\beta}(a := t)(s')(\mathtt{b}) = \begin{cases} val_{\mathscr{K},s,\beta}(t) & \textit{if}\ \mathtt{b} = \mathtt{a} \\ s'(\mathtt{b}) & \textit{otherwise} \end{cases}$$

$$\textit{for all}\ s' \in \mathscr{S},\ \mathtt{b} \in \mathsf{PVSym}$$

$$val_{\mathscr{K},s,\beta}(u_1 \,||\, u_2)(s') = val_{\mathscr{K},s,\beta}(u_2)(val_{\mathscr{K},s,\beta}(u_1)(s'))\ \textit{for all}\ s' \in \mathscr{S}$$

$$val_{\mathscr{K},s,\beta}(\{u_1\}u_2) = val_{\mathscr{K},s',\beta}(u_2),\ \textit{where}\ s' = val_{\mathscr{K},s,\beta}(u_1)(s)$$

$$val_{\mathscr{K},s,\beta}(\{u\}\phi) = val_{\mathscr{K},s',\beta}(\phi),\ \textit{where}\ s' = val_{\mathscr{K},s,\beta}(u)(s)$$

Definition 26 (Semantics of Java DL terms and formulas [2, Chapter 3])**.** *Let Prg be a Java program, $\mathscr{T}$ a type hierarchy for Prg, Σ a signature w.r.t. $\mathscr{T}$, $\mathscr{K} = (\mathscr{S}, \varrho)$ a Kripke structure for Σ, $s \in \mathscr{S}$ a state, and $\beta \colon \mathsf{VSym} \to D$ a variable assignment.*

For every Java DL term $t \in \mathsf{Trm}_A$, we define its evaluation by

$$val_{\mathscr{K},s,\beta}(t) = \mathsf{val}_{s,\beta}(t)\ \ ,$$

where $\mathsf{val}_{s,\beta}$ is defined as in the first-order case (Def. 21).

For every Java DL formula $\phi \in \mathsf{Fml}$, we define when ϕ is considered to be true with respect to $\mathscr{K}, s, \beta$, which is denoted with $(\mathscr{K}, s, \beta) \models \phi$, by Clauses 1–9 as shown in the definition of the semantics of FOL formulas (Def. 22) – with $\mathscr{M} = s$ and $(\mathscr{K}, s, \beta)$ replaced for $(\mathscr{M}, \beta)$ – in combination with the two new clauses:

10	$(\mathscr{K}, s, \beta) \models [\mathtt{p}]\phi$	*iff*	*there is no s' with $(s, s') \in \varrho(\mathtt{p})$ or* *$(\mathscr{K}, s', \beta) \models \phi$ for s' with $(s, s') \in \varrho(\mathtt{p})$*
11	$(\mathscr{K}, s, \beta) \models \langle\mathtt{p}\rangle\phi$	*iff*	*there is an s' with $(s, s') \in \varrho(\mathtt{p})$ and* *$(\mathscr{K}, s', \beta) \models \phi$ for s' with $(s, s') \in \varrho(\mathtt{p})$*

Finally, we define what it means for a Java DL formula to be valid or satisfiable. A first-order formula is satisfiable (resp. valid) if it holds in some (all) model(s) for some (all) variable assignment(s). Similarly, a Java DL formula is satisfiable (resp. valid) if it holds in some (all) state(s) of some (all) Kripke structure(s) $\mathscr{K}$ for some (all) variable assignment(s).

Definition 27. *[2, Chapter 3] Let Prg be a Java program, $\mathscr{T}$ a type hierarchy for Prg, Σ a signature w.r.t. $\mathscr{T}$, and $\phi \in \mathsf{Fml}$ a formula.*

ϕ is satisfiable *if there is a Kripke structure $\mathscr{K}$, a state $s \in \mathscr{S}$ and a variable assignment β such that $(\mathscr{K}, s, \beta) \models \phi$.*

ϕ is logically valid*, denoted by $\models \phi$, if $(\mathscr{K}, s, \beta) \models \phi$ for all Kripke structures $\mathscr{K}$, all states $s \in \mathscr{S}$ and all variable assignments β.*

,

Kripke structures can therefore be used to express the semantics of Java DL formulas.

2.3.3 Java Dynamic Logic Calculus

Java dynamic logic uses a *sequent calculus* [29] where rules are written in the form

$$\mathsf{ruleName}\ \frac{P_1 \dots P_n}{C}$$

The P_i are called the *premisses* and C the *conclusion* of the rule. If there are no premisses, the rule is called a *closing rule.* Premiss and conclusion contain the schematic variables Γ, Δ for sets of formulas, φ, ϕ for formulas and t, c for terms and constants. Γ, ϕ and φ, Δ stand for $\Gamma \cup \{\phi\}$ and $\{\varphi\} \cup \Delta$. An instance of a rule is obtained by consistently replacing the schematic variables in premiss and conclusion by the corresponding entities: sets of formulas, formulas, etc. Rule application works from the bottom up. Starting with an initial sequent, we apply a calculus rule leading to new sequents. This process is continued for each new sequent. Thus a proof tree is created. If all leaves of this proof tree have been closed, the proof is closed.

Definition 28. *A* proof tree *is a tree, shown with the root at the bottom, such that*

1. *each node is labeled with a sequent or the symbol $*$,*
2. *if an inner node n is annotated with $\Gamma \Longrightarrow \Delta$ then there is an instance of a rule whose conclusion is $\Gamma \Longrightarrow \Delta$ and the children nodes of n are labeled with the premisses of the rule instance, or the single child node is annotated with $*$ if this was a closing rule.*

A branch in a proof tree is called closed *if its leaf is labeled by $*$. A proof tree is called* closed *if all its branches are closed, or equivalently if all its leaves are labeled with $*$.*

A sequent $\Gamma \Longrightarrow \Delta$ can be derived if there is a closed proof tree whose root is labeled by $\Gamma \Longrightarrow \Delta$.

The calculus contains rules for first-order reasoning. The rules for conjunctions are shown as an example:

$$\mathsf{andLeft}\ \frac{\Gamma, \phi, \varphi \Longrightarrow \Delta}{\Gamma, \phi \wedge \varphi \Longrightarrow \Delta} \qquad \mathsf{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \varphi, \Delta}{\Gamma \Longrightarrow \phi \wedge \varphi, \Delta}$$

There are also the three closing rules:

$$\mathsf{close}\ \frac{}{\Gamma, \phi \Longrightarrow \phi, \Delta} \qquad \mathsf{closeTrue}\ \frac{}{\Gamma \Longrightarrow \mathsf{true}, \Delta} \qquad \mathsf{closeFalse}\ \frac{}{\Gamma, \mathsf{false} \Longrightarrow \Delta}$$

There are also rewrite rules to simplify terms and updates. In addition, Java dynamic logic also contains calculus rules for modalities and the programs within them. For example the calculus rule for the empty modality is:

$$\mathsf{emptyModality}\ \frac{\Gamma \Longrightarrow \{\mathscr{U}\}\phi, \Delta}{\Gamma \Longrightarrow \{\mathscr{U}\}[\ ]\phi, \Delta}$$

In general we are interested in modalities containing programs. The *symbolic execution* rules operate on the first *active* statement p in a modality $[\pi p \omega]$. The non-active prefix π consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method-frame(...){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active command is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately, and in order to access the value of the `this` object of the current method invocation and determine in which class the implementation resides, so as to correctly handle `super`, `private` and `static` calls.

The postfix ω denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on (in particular, ω contains closing braces corresponding to the opening braces in π). For example, if a rule is applied to the following Java block operating on its first active command "`i=0;`", then the non-active prefix π and the "rest" ω are the indicated parts of the block:

$$\underbrace{\texttt{l:\{try\{}}_{\pi}\ \texttt{i=0;}\ \underbrace{\texttt{j=0; \} finally\{ k=0; \}\}}}_{\omega}$$

Symbolic Execution Calculus Rules

Many symbolic execution calculus rules are concerned with splitting complex Java statements into multiple simpler statements. When a rule contains a schema expression se, this is a *simple expression* (a program variable or constant), while nse can be any expression, such as `o.a[o.a.length - x].f.m()`. Some rules to split complex statements are shown below, where T, T_0, T_1 are appropriate types for the expressions:

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ T\ \texttt{x = }nse_1\texttt{; x.f = }nse_2\texttt{;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ nse_1\texttt{.f = }nse_2\texttt{;}\ \omega]\varphi, \Delta}$$

$$\frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}se \doteq \texttt{null} \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{throw new NullPointerException();}\ \omega]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}se \neq \texttt{null} \Longrightarrow \{\mathcal{U}\}[\pi\ T_0\ \texttt{x}_0\texttt{ = }se\texttt{;}\ T_1\ \texttt{x}_1\texttt{ = }nse\texttt{; x}_0\texttt{.f = }x_1\texttt{;}\ \omega]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ se\texttt{.f = }nse\texttt{;}\ \omega]\varphi, \Delta}$$

Once we have reduced the complex statements to simple statements, there are calculus rules to perform these. We show here the calculus rules for assignment to a local variable, object field and primitive array index. The assignment rules move the assignment to the update. For assignments to fields and array indices this involves modifying the `heap`:

$$\textsf{assignLocalVariable}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\texttt{x} := \mathrm{se}\}[\pi\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{x = }se\texttt{;}\ \omega]\varphi, \Delta}$$

assignObjectField

$$\frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}\texttt{o} \doteq \texttt{null} \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{throw new NullPointerException();}\ \omega]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\texttt{o} \neq \texttt{null} \Longrightarrow \{\mathcal{U}\}\{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{o}, \texttt{f}, se)\}[\pi\ \omega]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{o.f = }se\texttt{;}\ \omega]\varphi, \Delta}$$

assignPrimitiveArrayIndex

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}\texttt{a} \doteq \texttt{null} \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{throw new NullPointerException();}\ \omega]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\texttt{a} \neq \texttt{null}, \{\mathcal{U}\}(se_1 < 0 \lor se_1 \geq \mathit{length}(\texttt{a})) \Longrightarrow \\ \qquad \{\mathcal{U}\}[\pi\ \texttt{throw new ArrayIndexOutOfBoundsException();}\ \omega]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\texttt{a} \neq \texttt{null}, \{\mathcal{U}\}se_1 \geq 0, \{\mathcal{U}\}(se_1 < \mathit{length}(\texttt{a})) \Longrightarrow \\ \qquad\qquad \{\mathcal{U}\}\{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{a}, \mathit{arr}(se_1), se_2)\}[\pi\ \omega]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{a[}se_1\texttt{] = }se_2\texttt{;}\ \omega]\varphi, \Delta}$$

Calculus Rules for Non-standard Control Flow

We consider the labeled `break` statement. The following rules allow the `break` to be propagated out of non-matching blocks and `try`-blocks, or resolved by leaving the labeled block.

blockBreakNoMatch

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{break}\ l';\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1 : \ldots l_n :\texttt{\{}\ \texttt{break}\ l';\ p\ \texttt{\}}\ \omega]\varphi, \Delta}\text{, if } \forall i \in \{1, \ldots, n\}.\ l' \neq l_i$$

$$\text{blockBreakLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1 : \ldots l_i : \ldots l_n :\texttt{\{}\ \texttt{break}\ l_i;\ p\ \texttt{\}}\ \omega]\varphi, \Delta}$$

$$\text{tryBreakLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ r\ \texttt{break}\ l';\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{try\{}\ \texttt{break}\ l';\ p\ \texttt{\}}\ cs\ \texttt{finally\{}\ r\ \texttt{\}}\omega]\varphi, \Delta}$$

It is important to point out that the contents r of the `finally`-block will be executed before the `break` statement is again the active statement.

Calculus Rules for Method Calls

The rules for preparation and execution of a method call and for returning from said call are given here. Before the method call is invoked, the target of the call must first be established.

$$\text{methodCallUnfoldTarget}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ T_{nse}\ v_0 = nse;\ \ lhs = v_0.mname(args);\ \ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ lhs = nse.mname(args);\ \ \omega]\varphi, \Delta}$$

Next the method's arguments are simplified, as Java uses pass-by-value.

methodCallUnfoldArguments

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ T_{e_{l_1}}\ v_1 = e_{l_1};\ \ldots;\ T_{e_{l_k}}\ v_k = e_{l_k};\ lhs = se.mname(a_1, \ldots, a_n);\ \ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ lhs = se.mname(e_1, \ldots, e_n);\ \ \omega]\varphi, \Delta}$$

Where a_i is either e_i or v_{l_i}, depending on whether e_i is a simple expression or not.

As Java allows for inheriting, the actual method implementation to be called is not a function merely of the method's name. Rather, the method's name, signature and type of the target influence in which class the implementation is located. The rule methodCall splits on the actual type of the target (including the premiss for type *Null*), annotating the method call with the implementation containing class C_i. Methods declared `static` or `private`, and `super` invocations can only have one implementing class and so the rule can be somewhat simplified. Furthermore, fresh program variables are provided for the method parameters, as modifications to the parameters do not leave the method in Java due to pass-by-value.

methodCall

$$\frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}se \doteq \texttt{null} \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{throw new NullPointerException();}\ \omega]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}se \neq \texttt{null} \Longrightarrow \{\mathcal{U}\}[\pi\ T_{se_1}\ p_1 = se_1; \ldots T_{se_n}\ p_n = se_n;\ \ T_{lhs}\ v_0;\ \ ifCscd;\ \ lhs = v_0;\ \ \omega]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ lhs = se.mname(se_1, \ldots, se_n);\ \ \omega]\varphi, \Delta}$$

Where *ifCscd* is ...

```
if ($se$ instanceof $C_1$)
    $\mathit{v}_0 = se.\mathit{mname}(p_1, \ldots, p_n)@C_1;$
else if ($se$ instanceof $C_2$)
    $\mathit{v}_0 = se.\mathit{mname}(p_1, \ldots, p_n)@C_2;$
    ⋮
else if($se$ instanceof $C_{k-1}$)
    $\mathit{v}_0 = se.\mathit{mname}(p_1, \ldots, p_n)@C_{k-1};$
else $\mathit{v}_0 = se.\mathit{mname}(p_1, \ldots, p_n)@C_k;$
```

It is important to note that in contrast to the remaining calculus rules, methodCall requires a closed-world assumption when applied to non-`private` instance methods. This is because *ifCscd* must list *all* method implementations which could be called and this is not known in an open world.

The execution of a non-recursive method can be handled by simply expanding the method body. This creates a new method-frame containing the method identifier, `this` reference for calls directly within this method-frame, and program variable to store the result to. Static methods do not have a `this` reference and `void` methods do not contain a program variable for the result.

methodBodyExpand

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{method-frame(source=}m\texttt{, result->}\mathit{lhs}\texttt{, this=}se\texttt{)}\ :\ \{\ \mathit{body}\ \}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathit{lhs} = se.\mathit{mname}(\mathit{v}_1, \ldots, \mathit{v}_n)@\mathit{Class}; \omega]\varphi, \Delta}$$

Where m is a method identifier `mname(`sig`)@Class`, with sig the signature $\mathit{Class} \times T_{\mathit{fp}_1} \times \cdots \times T_{\mathit{fp}_n} \to T$ (including implicit `this` parameter and return value) of the method *mname* as declared in class *Class* with the formal parameters fp_i.

Return statements at method-frames cause the result value for the method to be set and the beginning of method-frame cleanup to be triggered.

methodCallReturn

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{method-frame(source=}m\texttt{, this=}se\texttt{)}\ :\ \{\ \mathit{v} = se;\ \}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{method-frame(source=}m\texttt{, result->}\mathit{v}\texttt{, this=}se\texttt{)}\ :\ \{\ \texttt{return}\ se;\ p\ \}\ \omega]\varphi, \Delta}$$

Note that the `result` pointer has disappeared from the method-frame, making it easy to check when the following rule is applicable (method was declared `void` or return value already set):

methodCallEmpty
$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{method-frame(source=}m\texttt{, this=}se\texttt{)}\ :\ \{\ \}\ \omega]\varphi, \Delta}$$

The following rule can be used for methods declared `void` which have a `return` statement:

methodCallEmptyReturn

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{method-frame(source=}m\texttt{, this=}se\texttt{)}\ :\ \{\ \texttt{return;}\ \texttt{p}\ \}\ \omega]\varphi, \Delta}$$

Method-frames can also be left exceptionally:

methodCallThrow

$$\frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{throw}\ se;\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{method-frame(source=}m\texttt{, result->}\mathit{v}\texttt{, this=}se\texttt{)}\ :\ \{\ \texttt{throw}\ se;\ \texttt{p}\ \}\ \omega]\varphi, \Delta}$$

The general rules for `return` and `throw` statements are to first evaluate the argument and then propagate the statement out of all (unlabeled, labeled, `try`-) blocks until it reaches the method-frame. The `catch` blocks can resolve matching `throw` statements, while statements in `finally`-blocks have precedence over propagation.

For more information on Java dynamic logic and the explicit heap notation see [9, 63, 2]

2.4 Abstracting Program States

The combination of abstract interpretation with dynamic logic presented in [12] introduces a logical representation of abstract domains, in the form of partially interpreted γ- and χ-symbols.

Definition 29 (Logical Representation of Abstract Domains)**.** *For an abstract domain $(A, \sqcup, \sqsubseteq)$, the signature Σ_A provides for every $a \in A$:*

1. *a unary predicate symbol χ_a*
2. *an infinite number of constant symbols $\gamma_{a,z}$, where $z \in \mathbb{Z}$*

For this signature Σ_A only interpretations I satisfying the following are considered:

1. *$I(\chi_a) = \gamma(a)$, for all $a \in A$*
2. *$I(\gamma_{a,z}) \in \gamma(a)$, for all $a \in A, z \in \mathbb{Z}$*

This ensures that there exist *characteristic functions* χ_a and constrained values $\gamma_{a,z}$ for the abstract elements of A.

Definition 30 ((P, C)-weaker)**.** *Let P be a sequent proof, C a set of constraints and $\mathcal{U}_1, \mathcal{U}_2$ updates. $\mathcal{U}_2$ is said to be (P, C)-*weaker *than $\mathcal{U}_1$, if for all interpretations I, states s and variable assignments β, where for all $\phi \in C$ it holds that $val_{I,s,\beta}(\phi) = tt$, the following holds:*

$$val_{I,s,\beta}(\mathcal{U}_1) \in \{val_{I',s,\beta}(\mathcal{U}_2) \mid I \simeq_{P,C} I'\}$$

where $I \simeq_{P,C} I'$ means that I and I' coincide on all function and predicate symbols occurring in P or C.

For any valid semantics of the update $\mathcal{U}_1$ under the set of constraints C, the (P, C)-weaker update $\mathcal{U}_2$ can be given the same semantics. Additionally, $\mathcal{U}_2$ can possibly be given semantics which $\mathcal{U}_1$ cannot.

Definition 31 (Update Join)**.** *An* update join $\dot{\sqcup}$ *has the signature*

$$\dot{\sqcup} : (2^{\mathsf{Fml}} \times \mathsf{Upd}) \times (2^{\mathsf{Fml}} \times \mathsf{Upd}) \to \mathsf{Upd}$$

and fulfills the following properties:

1. *The result of $(\mathcal{U}_1, C_1) \,\dot{\sqcup}\, (\mathcal{U}_2, C_2)$ is (P, C)-weaker than $(\mathcal{U}_1, C_1)$*
2. *The result of $(\mathcal{U}_1, C_1) \,\dot{\sqcup}\, (\mathcal{U}_2, C_2)$ is (P, C)-weaker than $(\mathcal{U}_2, C_2)$*

The result of an *update join* is an update which can express at least all possible values either of its inputs can express. A concrete implementation of an update join is given in [12]

Definition 32 (Substitution)**.** *A syntactic* substitution $[\cdot/\cdot]$ *applied to a formula ϕ, term t or set of constraints C is to be understood as follows:*

- *$\phi[x/y]$ is the formula resulting by syntactically replacing every occurrence of x in ϕ with y*
- *$t[x/y]$ is the term resulting by syntactically replacing every occurrence of x in t with y*
- *$C[x/y] := \{\phi[x/y] \mid \phi \in C\}$*

Definition 33 (Vector Notation)**.** *Vectors $(x_1, \ldots, x_n)$ are abbreviated $\bar{x}$. Operations on vectors are to be understood as follows:*

- $\bar{t} \doteq \bar{t}' := t_1 \doteq t'_1 \wedge \ldots \wedge t_n \doteq t'_n$
- $\chi_{\bar{a}}(\bar{t}) := \chi_{a_1}(t_1) \wedge \ldots \wedge \chi_{a_n}(t_n)$
- $\exists \bar{y}.\phi := \exists y_1. \cdots \exists y_m. \ \phi$
- *$t[\bar{x}/\bar{y}]$ and $\phi[\bar{x}/\bar{y}]$ are the result of simultaneously syntactically replacing every occurrence of x_i with y_i for all $i \in \{1, \ldots, n\}$ in the term t or formula ϕ*

Generating Invariant Updates: Abstract Program States Expressing Loop Invariants

The ability to express abstract values syntactically via γ-symbols and join updates containing both concrete and abstract values, allows the fixed point algorithm to find an abstract update weaker than the initial update and all updates reachable by symbolic execution of a loop for any number of iterations. Based on the proof P and initial sequent *seq* encountering a loop, where *seq* is

$$\Gamma \Longrightarrow \{\mathscr{U}\}[\mathtt{while\ (}g\mathtt{)\ \{\ p\ \};\ r}]\phi, \Delta$$

this is accomplished in the following manner:

1. Set $\mathscr{U}'$ to $\mathscr{U}$, C to $\Gamma \cup !\Delta$.
2. Unroll the loop in $C \Longrightarrow \{\mathscr{U}'\}[\mathtt{while\ (}g\mathtt{)\ \{\ p\ \};\ r}]\phi$ and perform symbolic execution of one iteration, gathering the n sequents $\Gamma_i \Longrightarrow \{\mathscr{U}_i\}[\mathtt{while\ (}g\mathtt{)\ \{\ p\ \};\ r}]\phi, \Delta_i$, for $1 \leq i \leq n$ of open branches leading back to the loop entry.
3. Use update joining to calculate $\mathscr{U}^* = (C, \mathscr{U}') \dot{\sqcup} (\Gamma_1 \cup !\Delta_1, \mathscr{U}_1) \dot{\sqcup} \cdots \dot{\sqcup} (\Gamma_n \cup !\Delta_n, \mathscr{U}_n)$.
4. If $\mathscr{U}'$ is (P, C)-weaker than $\mathscr{U}^*$, a fixed point has been found and $\mathscr{U}'$ expresses an *invariant update.*
5. Otherwise, set $\mathscr{U}'$ to $\mathscr{U}^*$ and go to step 2.

Applying Invariant Updates

An invariant update may be used in the Java dynamic logic calculus rule invariantUpdate presented in [12], where $\bar{\mathtt{x}}$ is a duplicate free vector of all program variables assigned to in $\mathscr{U}$ or $\mathscr{U}'$, $\bar{c}$ is a vector of the same length as $\bar{\mathtt{x}}$ containing fresh constant symbols, $\bar{\gamma}$ is a vector of all γ-symbols introduced in $\mathscr{U}'$ and $\bar{y}$ is a vector of the same length as $\bar{\gamma}$ containing fresh logical variables:

$$\textsf{invariantUpdate}\ \frac{\begin{array}{l}\Gamma, \{\mathscr{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{y}.(\chi_{\bar{a}}(\bar{y}) \wedge (\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}))[\bar{\gamma}/\bar{y}]), \Delta \\ \Gamma, \{\mathscr{U}'\}g,\ \{\mathscr{U}'\}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathscr{U}'\}\neg g \Longrightarrow \{\mathscr{U}'\}[\mathtt{r}]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathscr{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};\ r}]\varphi, \Delta}$$

The calculus rule invariantUpdate has three premisses, ensuring that:

1. the update $\mathscr{U}'$ is weaker than the initial update $\mathscr{U}$, i.e. that $\mathscr{U}'$ can express all possible program states upon loop entry;
2. the loop body preserves the update $\mathscr{U}'$, i.e. that $\mathscr{U}'$ can express all possible program states resulting from a single loop iteration starting from a program state expressible by $\mathscr{U}'$; and finally
3. the use case is valid, i.e. that evaluating the remaining program after the loop in program states expressible by $\mathscr{U}'$ results in provable sequents.

3 Abstract Domains for Local Variable Types

The first step in extending the ideas put forward in [12] from a toy language to a closer approximation of full Java programs is to make sure our abstractions can cover all types occurring in updates. In this chapter we describe the abstract domains chosen for the various types a local program variable can have, where "local program variable" includes method parameters and return values. In the next chapter we describe an abstract domain for program heaps, to allow abstract values for the program variable `heap` expressing the state of the Java program's heap.

Before describing our chosen abstract domains, we briefly list what sort of abstractions we do not supply:

Relational abstract domains: While relational abstract domains are much better at expressing certain properties and have been used, for example, to express complicated numerical synergies between various variables, such as through congruence relations, polyhedra or linear or polynomial equations, they cannot easily be integrated with the concept of γ-symbols, as these rely on the immutability of any references values and can therefore not express something like "greater than the value of variable `x`" as this value could change. We do, however, offer a way to express some relational aspects when discussing helpful invariants in Chapter 5.

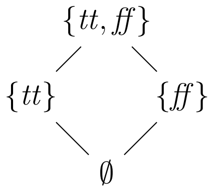
Abstract domains for floating point numbers: While floating point numbers are used in many calculations and are of course themselves abstractions of real numbers, they are rather hard to reason about in many ways and for the most part have been ignored in KeY. For this reason creating an abstract domain for floating point numbers is simply out of the scope of this thesis.

3.1 Abstract Domains for Primitive Types

The main primitive types in Java are various integer types and `boolean`. Java DL uses the *int* type domain for `byte`, `char`, `short` and `long`, and the *Boolean* type domain for `boolean`, while floating point numbers are not allowed. The semantics of any integer type is mapped to $\mathbb{Z}$. In order to be sound with regard to implementation of integers in Java, we can constrain the various integer types to the range of actual values in $\mathbb{Z}$ that they can have while ensuring Java integer operators are performed in symbolic execution rules as defined by the Java Language Specification [32], including numeric promotion, widening, arithmetic overflows and casts. On a theoretical level, however, often we wish to simplify and treat built-in arithmetic operators as their corresponding operators on $\mathbb{Z}$ ignoring integer overflow, etc. We offer abstract domains useful for both of these cases, as while the second option is known to be unsound when considering actual Java programs, it can be quite useful when examining a program expressing a theoretical algorithm on the full integer range $\mathbb{Z}$.

3.1.1 Abstract Domain for Booleans

We use the simple domain $\mathscr{A}^{\mathtt{bool}} = (2^{D^{\mathtt{bool}}}, \cup, \subseteq)$, with abstraction and concretization functions defined as simple identities $\alpha^{\mathtt{bool}}(X) = \gamma^{\mathtt{bool}}(X) = X$, shown in the following lattice:



We begin with abstracting integers assuming that all values in $\mathbb{Z}$ are possible and normal integer arithmetic rules are present. This makes abstractions such as a sign domain quite useful, as we know, for example, that two positive numbers added or multiplied result in a positive number. Two relatively simple abstract domains for $\mathbb{Z}$ with this in mind are given: The first is a finite width, finite height abstract domain extending somewhat on the sign lattices of [15, 12, 24]:

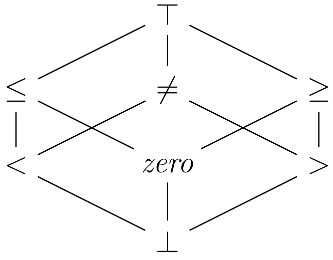


Figure 3.1: Abstract domain $\mathscr{A}^{sign}$

$\mathscr{A}^{sign} = (\{\top, \bot, zero, >, <, \geq, \leq, \neq\}, \sqcup_{sign}, \sqsubseteq_{sign})$, where $\sqcup_{sign}$ and $\sqsubseteq_{sign}$ can be inferred from the lattice in Figure 3.1 while concretization and abstraction functions are shown in Figure 3.2.

$$\begin{aligned}
\gamma^{sign}(\bot) &= \emptyset \\
\gamma^{sign}(zero) &= \{0\} \\
\gamma^{sign}(>) &= \{z \in \mathbb{Z} \mid z > 0\} \\
\gamma^{sign}(<) &= \{z \in \mathbb{Z} \mid z < 0\} \\
\gamma^{sign}(\neq) &= \mathbb{Z} \setminus \{0\} \\
\gamma^{sign}(\geq) &= \{z \in \mathbb{Z} \mid z \geq 0\} \\
\gamma^{sign}(\leq) &= \{z \in \mathbb{Z} \mid z \leq 0\} \\
\gamma^{sign}(\top) &= \mathbb{Z}
\end{aligned}
\qquad
\alpha^{sign}(X) = \begin{cases}
\bot & \text{, if } X = \emptyset \\
zero & \text{, if } X = \{0\} \\
> & \text{, if } X \neq \emptyset \land \forall x \in X.\ x > 0 \\
< & \text{, if } X \neq \emptyset \land \forall x \in X.\ x < 0 \\
\neq & \text{, if } 0 \notin X \land \exists x, y \in X.\ x > 0 \land y < 0 \\
\geq & \text{, if } 0 \in X \land X \neq \{0\} \land \forall x \in X.\ x \geq 0 \\
\leq & \text{, if } 0 \in X \land X \neq \{0\} \land \forall x \in X.\ x \leq 0 \\
\top & \text{, otherwise}
\end{cases}$$

Figure 3.2: Abstraction and concretization functions for abstract domain $\mathscr{A}^{sign}$

The second abstract domain expresses within which interval (lower bound, upper bound) the values reside. This *interval abstraction* was first proposed in [14] and provides much higher precision than the sign abstraction above, however the interval abstract domain has infinite height (and width) and therefore requires a widening operator.

The interval abstract domain is defined as $\mathscr{A}^{interval} = (A^{interval}, \sqcup_{interval}, \sqsubseteq_{interval})$, where

$$\begin{aligned}
A^{interval} &= \{\bot\} \cup \{(x, y) \mid x \in (\mathbb{Z} \cup \{-\infty\}) \land y \in (\mathbb{Z} \cup \{\infty\}) \land x \leq y\} \\
\forall a \in A^{interval}.\ \bot &\sqsubseteq_{interval} a \\
(x_1, y_1) \sqsubseteq_{interval} (x_2, y_2) &= (x_1 \geq x_2 \land y_1 \leq y_2) \\
\forall a \in A^{interval}.\ \bot \sqcup a &= a \sqcup \bot = a \\
(x_1, y_1) \sqcup_{interval} (x_2, y_2) &= (min(x_1, x_2), max(y_1, y_2))
\end{aligned}$$

An excerpt of the abstract domain lattice for $\mathscr{A}^{interval}$ is shown in Figure 3.3, where dots and dotted lines represent an infinite number of abstract elements. Abstraction and concretization functions are shown in Figure 3.4.

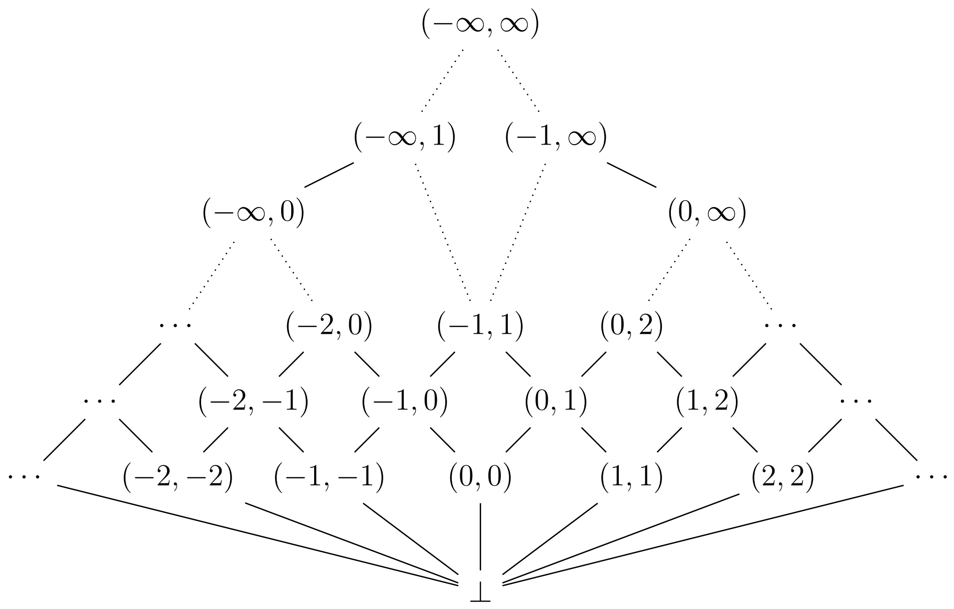


Figure 3.3: Excerpt of abstract domain $\mathscr{A}^{interval}$

$$
\begin{aligned}
\alpha^{interval}(M) &= \begin{cases} \perp & \text{, if } M = \emptyset \\ (\inf(M), \sup(M)) & \text{, if infimum and supremum of } M \text{ exists} \\ (\inf(M), \infty) & \text{, if infimum of } M \text{ exists but supremum does not} \\ (-\infty, \sup(M)) & \text{, if supremum of } M \text{ exists but infimum does not} \\ (-\infty, \infty) & \text{, if neither infimum nor supremum of } M \text{ exist} \end{cases} \\
\gamma^{interval}(\perp) &= \emptyset \\
\gamma^{interval}((x, y)) &= \{z \in \mathbb{Z} \mid x \leq z \leq y\}
\end{aligned}
$$

Figure 3.4: Abstraction and concretization functions between $\mathscr{A}^{interval}$ and $\mathbb{Z}$

The widening operator $\nabla_{interval}$ is defined as follows:

$$
\begin{aligned}
\nabla_{interval} : A^{interval} \times A^{interval} &\to A^{interval} \\
(\perp, i) &\mapsto i \\
(i, \perp) &\mapsto i \\
((x_1, y_1), (x_2, y_2)) &\mapsto (x_3, y_3), \text{ where } x_3 = x_1 \text{ if } x_1 \leq x_2,\ x_3 = -\infty \text{ otherwise} \\
&\qquad\qquad\quad \text{and } y_3 = y_1 \text{ if } y_1 \geq y_2,\ y_3 = \infty \text{ otherwise}
\end{aligned}
$$

We introduce one further abstract domain for integers, as the standard integer abstract domain to be used when Java integer semantics for arithmetic operators is enforced. The important points here are as follows:

- Only a finite subset of $\mathbb{Z}$ is now relevant, as values outside of the range defined by the variable type are not permissible. As we still map to and from the full concrete domain of $\mathbb{Z}$, we still need to take values outside of this range into account, but this can simply be done by having these map to $\top$.

- Intervals are still useful, but as most integer types are signed and expressed in two's complement, incrementing the "bigger" number might lead to a smaller, negative number. An abstract domain for these *wrapped intervals* has been presented in [49], but they use a non-lattice abstract domain which causes problems in the strict definition of a join for non-overlapping intervals [28].

As a Java `long` is the integer type which can contain the most values, we can limit the relevant range for integers to those between -2^{63} and $2^{63}-1$. In order to take overflows into account and still be able to consider intervals, we let an abstract element combine two finite intervals in $\mathbb{Z}$: an interval containing negative numbers and one containing non-negative numbers.

The abstract domain $\mathscr{A}^{int} = (A^{int}, \sqcup_{int}, \sqsubseteq_{int})$, where:

$$\begin{aligned} A^{int} = \{\top\} \cup \{(n,p) \mid\ & n \in (\{\bot\} \cup \{(x,y) \mid x,y \in \mathbb{Z}.\ -2^{63} \leq x \leq y < 0\}) \\ & \text{and } p \in (\{\bot\} \cup \{(x,y) \mid x,y \in \mathbb{Z}.\ 0 \leq x \leq y < 2^{63}\})\} \end{aligned}$$

Joining and ordering is done by joining or checking both intervals:

$$\begin{aligned} &\top \sqcup_{int} a = a \sqcup_{int} \top = \top \\ &(n_1,p_1) \sqcup_{int} (n_2,p_2) = (n_1 \sqcup'_{int} n_2, p_1 \sqcup'_{int} p_2) \\ &\bot \sqcup'_{int} b = b \sqcup'_{int} \bot = b \\ &(x_1,y_1) \sqcup'_{int} (x_2,y_2) = (min(x_1,x_2), max(y_1,y_2)) \end{aligned}$$

$$a_1 \sqsubseteq_{int} a_2 = \begin{cases} tt & \text{, if } a_2 = \top \\ f\!f & \text{, if } a_1 = \top \text{ and } a_2 \neq \top \\ n_1 \sqsubseteq'_{int} n_2 \wedge p_1 \sqsubseteq'_{int} p_2 & \text{, otherwise, where } a_1 = (n_1,p_1) \text{ and } a_2 = (n_2,p_2) \end{cases}$$

$$b_1 \sqsubseteq'_{int} b_2 = \begin{cases} tt & \text{, if } b_1 = \bot \\ f\!f & \text{, if } b_2 = \bot \text{ and } b_1 \neq \bot \\ x_2 \leq x_1 \wedge y_1 \leq y_2 & \text{, otherwise, where } b_1 = (x_1,y_1) \text{ and } b_2 = (x_2,y_2) \end{cases}$$

An excerpt of the abstract domain is shown in Figure 3.5 with dots and dotted lines representing a finite number of further abstract elements. Abstraction and concretization functions are shown in Figure 3.6.

The abstract domain $\mathscr{A}^{int}$ has a lattice of finite height and therefore does not require a widening operator to ensure termination when finding a fixed point. However, as the height of the lattice is quite large ($2^{64}+2$), supplying a widening operator is still useful. As in general higher precision is more important for numbers closer to zero than it is for numbers farther away (a fact that is used in the IEEE definition of floating point numbers), our widening operator should widen based on a logarithmic scale. As all numbers in Java are stored as a certain number of bits, it is fairly straightforward to choose 2 as the logarithmic base. In addition to widening to a certain power of two, it can also often make sense to widen to one less or one more than the power of two, for example to reach 127, the upper bound for a `byte`. Combining these ideas we define our widening operator as follows:

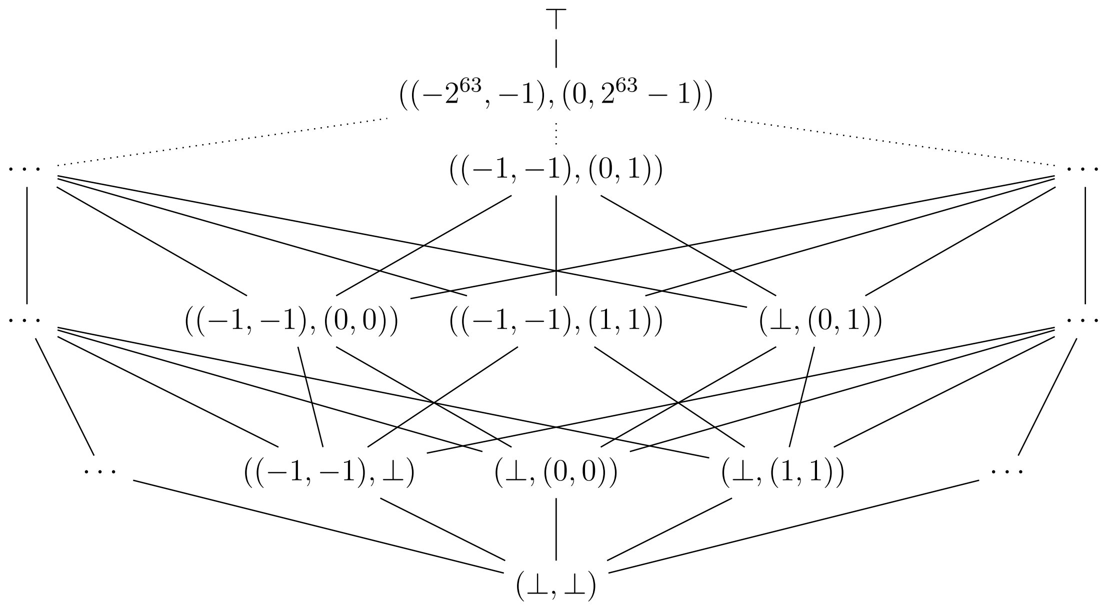


Figure 3.5: Excerpt of abstract domain $\mathscr{A}^{int}$

$$
\begin{aligned}
\alpha^{int}(M) &= \begin{cases} \top & \text{, if } \exists m \in M.\ m < -2^{63} \vee m \geq 2^{63} \\ (\alpha'_{int}(N), \alpha'_{int}(P)) & \text{, otherwise, where } N = \{m \in M \mid m < 0\} \\ & \text{ and } P = \{m \in M \mid m \geq 0\} \end{cases} \\
\alpha'_{int}(M) &= \begin{cases} \bot & \text{, if } M = \emptyset \\ (min(M), max(M)) & \text{, otherwise} \end{cases} \\
\gamma^{int}(\top) &= \mathbb{Z} \\
\gamma^{int}((\bot,\bot)) &= \emptyset \\
\gamma^{int}(((x,y),\bot)) &= \{z \in \mathbb{Z} \mid x \leq z \leq y\} \\
\gamma^{int}((\bot,(x,y))) &= \{z \in \mathbb{Z} \mid x \leq z \leq y\} \\
\gamma^{int}(((x_1,y_1),(x_2,y_2))) &= \{z \in \mathbb{Z} \mid (x_1 \leq z \leq y_1) \vee (x_2 \leq z \leq y_2)\}
\end{aligned}
$$

Figure 3.6: Abstraction and concretization functions between $\mathscr{A}^{int}$ and $\mathbb{Z}$

$$\begin{aligned}\nabla_{int} : A^{int} \times A^{int} &\to A^{int}\\ (\top, a) &\mapsto \top\\ (a, \top) &\mapsto \top\\ ((n_1, p_1), (n_2, p_2)) &\mapsto (W^n_{int}(n_1, n_2), W^p_{int}(p_1, p_2)), \text{ where:}\end{aligned}$$

$$\begin{aligned}W^n_{int}(\perp, b) &= b\\ W^n_{int}(b, \perp) &= b\\ W^n_{int}((x_1, y_1), (x_2, y_2)) &= (x_3, y_3), \text{ where } x_3 = x_1 \text{ if } x_1 \leq x_2,\ x_3 = -1 * next(-1 * x_2) \text{ otherwise}\\ &\qquad\qquad\qquad \text{and } y_3 = y_1 \text{ if } y_1 \leq y_2,\ y_3 = -1 * prev(-1 * x_2) \text{ otherwise}\end{aligned}$$

$$\begin{aligned}W^p_{int}(\perp, b) &= b\\ W^p_{int}(b, \perp) &= b\\ W^p_{int}((x_1, y_1), (x_2, y_2)) &= (x_3, y_3), \text{ where } x_3 = x_1 \text{ if } x_1 \leq x_2,\ x_3 = prev(x_2) \text{ otherwise}\\ &\qquad\qquad\qquad \text{and } y_3 = y_1 \text{ if } y_1 \leq y_2,\ y_3 = next(x_2) \text{ otherwise}\end{aligned}$$

$$next(x) = \begin{cases} x & \text{, if } \exists n \in \mathbb{N}.\ x = 2^n - 1 \lor x = 2^n \lor x = 2^n + 1\\ 2^n - 1 & \text{, otherwise, where } n \text{ is chosen such that } 2^{n-1} < x < 2^n\end{cases}$$

$$prev(x) = \begin{cases} x & \text{, if } \exists n \in \mathbb{N}.\ x = 2^n - 1 \lor x = 2^n \lor x = 2^n + 1\\ 2^n + 1 & \text{, otherwise, where } n \text{ is chosen such that } 2^n < x < 2^{n+1}\end{cases}$$

3.2 Abstract Domain for Objects

Most of the information about objects actually resides on the heap. There are only a few things we can say about objects directly, such as whether or not two objects are equal. We do not use relational abstract domains directly, but there is one object which always exists and we always have reference to: `null`. Additionally, we can abstract objects based on the value the *length* function returns for them. Furthermore, each object is of precisely one dynamic type, allowing abstraction based on types.

We therefore first introduce abstract domains for objects based on each of these points separately and can then combine them into one abstract domain for objects $\mathscr{A}^{Object}$.

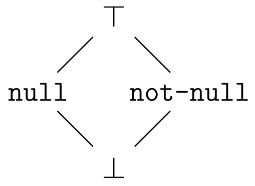
3.2.1 Null/Not-null Abstract Domain

The abstract domain $\mathscr{A}^{Object}_{\texttt{null}}$ for objects based on reference equality to `null` is quite simple and at the same time incredibly useful, in that it can be used to check for possible `NullPointerExceptions` or prove the lack thereof in a piece of Java code.

$\mathscr{A}^{Object}_{\texttt{null}}$ is shown in Figure 3.7 with abstraction and concretization functions.

3.2.2 Length Abstract Domain

An abstract domain for objects based on the result of the built-in *length* function is useful only for arrays. For all other object types this value is some arbitrary natural number which has no further meaning [63, pages 84-85]. For arrays, however, abstracting these to their length can be quite helpful, for example in



$$\alpha_{\mathscr{A}^{Object}_{\texttt{null}}}(X) = \begin{cases} \bot & \text{, if } X = \emptyset \\ \texttt{null} & \text{, if } X = \{null\} \\ \texttt{not-null} & \text{, if } null \notin X \\ \top & \text{, otherwise} \end{cases}$$

$$\gamma_{\mathscr{A}^{Object}_{\texttt{null}}}(x) = \begin{cases} \emptyset & \text{, if } x = \bot \\ \{null\} & \text{, if } x = \texttt{null} \\ D^{Object} \setminus \{null\} & \text{, if } x = \texttt{not-null} \\ D^{Object} & \text{, if } x = \top \end{cases}$$

Figure 3.7: Abstract Domain $\mathscr{A}^{Object}_{\texttt{null}}$

order to prove no `ArrayIndexOutOfBoundsException` is thrown. Furthermore, one could conclude based on this abstraction whether a loop iterating over an array should be unrolled completely or a loop invariant generated for it. A large fraction of all loops merely iterate over a structure such as an array, such that this, coupled with knowledge of the array's length, could be used to drastically improve certain proofs.

We require an abstract domain $\mathscr{A}^{\mathbb{Z}}$ for the concrete domain $\mathbb{Z}$ and map each object's length to said abstract domain. We can then define the abstract domain for objects based on their *length* as $\mathscr{A}^{Object}_{\texttt{length}} := \mathscr{A}^{\mathbb{Z}}$ with abstraction and concretization functions as follows:

$$\alpha_{\mathscr{A}^{Object}_{\texttt{length}}}(X) = \alpha_{\mathscr{A}^{\mathbb{Z}}}(\{length^{\mathscr{M}}(x) \mid x \in X\})$$

$$\gamma_{\mathscr{A}^{Object}_{\texttt{length}}}(x) = \{o \in D^{Object} \mid length^{\mathscr{M}}(o) \in \gamma_{\mathscr{A}^{\mathbb{Z}}}(x)\}$$

We can use any abstract domain for $\mathbb{Z}$, for example the sign domain in Figure 3.1. However, using this abstract domain would not be very clever as the abstract elements $<$ and $\leq$ will never abstract valid array lengths, while the abstraction of both 1 and 10000 to the same abstract element $>$ is not very helpful. Instead, let us consider the following points:

1. Iterating over an array of length 0 is trivial (do not enter loop) and therefore full precision should be kept, rather than abstracting by applying a loop invariant.

2. Iterating over an array of length 1 is similarly trivial (execute the loop body once) and therefore full precision should also be kept here by unrolling the loop, rather than applying a loop invariant. The loop invariant rule must still prove that the loop body preserves the invariant, thus execution of the loop body is always required once, even when applying a loop invariant.

3. Iterating over an array of length 2 or 3 can usually be done reasonably quickly by unrolling the loop a sufficient number of times, therefore unrolling should be favored over applying a loop invariant except in cases where symbolic execution of the loop body is extremely costly.

4. Iterating over an array of length 4 or 5 can often be done reasonably quickly by loop unrolling, therefore applying a loop invariant should only be done for somewhat complex loop bodies.

5. Iterating over an array of length 6 to 10, applying a loop invariant should be favored, except in cases where the loop body is trivial.

6. Iterating over an array of length greater than 10 should almost always be solved by applying a loop invariant.

The above are reasonable guidelines (or in the case of lengths 0 and 1 simple facts), such that we can present the abstract domain in Figure 3.8 for the concrete domain $\mathbb{Z}$ and therefore also for objects based on their length.

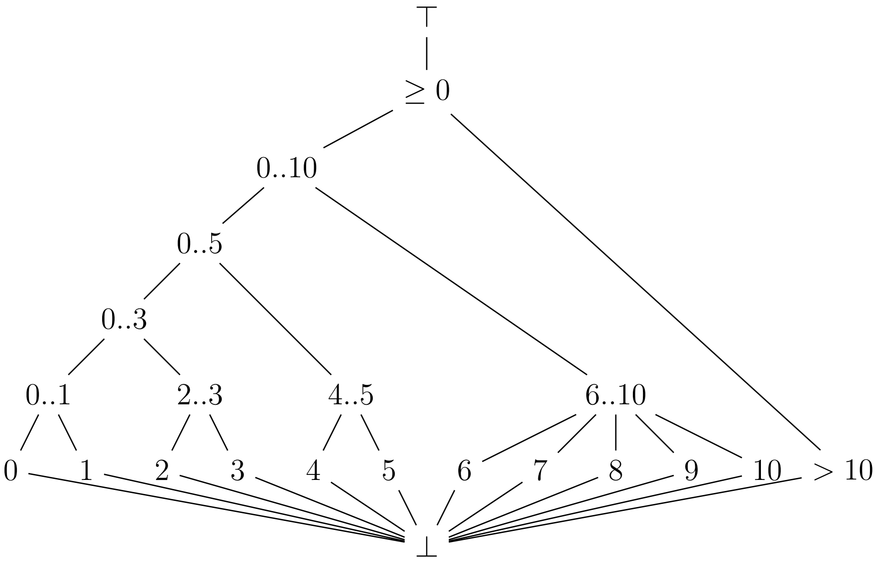


Figure 3.8: An Abstract Domain $\mathscr{A}^{Object}_{\texttt{length}}$

3.2.3 Type Abstract Domain

Abstracting on object type requires knowledge of the type hierarchy. However, due to logical consequence of a formula requiring that the formula hold in all extensions of the type hierarchy, we must in essence create an abstract domain based on not just the type hierarchy given directly by the program, but any extension thereof.

For a set of objects X we offer abstractions for their types based on which exact types are present in X, i.e. a set of types such that each element in X is an exact instance of one of those types.

For any given type hierarchy $\mathscr{T}$ we must create an abstract domain, such that there exist abstraction and concretization functions for all type hierarchies $\mathscr{T}'$ which extend $\mathscr{T}$.

For a given type hierarchy $\mathscr{T} = (\mathsf{TSym}, \sqsubseteq)$ we first split the set of types TSym into the disjoint sets TSym_a and TSym_d, where TSym_a contains all *abstract* types for Java classes declared `abstract` and for `interfaces`, as well as the empty type $\bot$. TSym_d contains all other types, the so-called *dynamic* types. We now define the set of all dynamic object types $O_d = \{T \in \mathsf{TSym}_d \mid T \sqsubseteq \mathit{Object}\}$ and based on this define the abstract domain $\mathscr{A}^{Object}_{O_d}$, as shown in Figure 3.9. Then for any type hierarchy extension $\mathscr{T}' = (\mathsf{TSym}', \sqsubseteq')$ of $\mathscr{T}$ the abstraction and concretization functions are given in Figure 3.10.

The abstract domains $\mathscr{A}^{Object}_{O_d}$ can be used, for example, to:

- prove that casting of an object does not cause a `ClassCastException` to be thrown,
- prove that no `ArrayStoreException` is thrown when inserting an object into an array,
- prove that an `instanceof` check will be successful,
- prove that an `instanceof` check will be unsuccessful, and/or
- narrow the list of possible method body instantiations down which is created when unfolding a method call.

$$
\begin{aligned}
\mathscr{A}^{Object}_{O_d} &= (A^{Object}_{O_d}, \sqsubseteq^{Object}_{O_d}, \sqcup^{Object}_{O_d}) \\
A^{Object}_{O_d} &= \{\top\} \cup (2^{O_d} \setminus O_d) \\
X \sqcup^{Object}_{O_d} Y &= \begin{cases} \top & \text{, if } X = \top \text{ or } Y = \top \text{ or } X \cup Y = O_d \\ X \cup Y & \text{, otherwise} \end{cases} \\
X \sqsubseteq^{Object}_{O_d} Y &= \begin{cases} tt & \text{, if } Y = \top \\ ff & \text{, if } X = \top \text{ and } Y \neq \top \\ X \subseteq Y & \text{, otherwise} \end{cases}
\end{aligned}
$$

Figure 3.9: Family of Abstract Domains $\mathscr{A}^{Object}_{O_d}$

$$
\begin{aligned}
\alpha^{Object}_{O_d,\mathscr{T}'}(X) &= \begin{cases} \top & \text{, if } \exists x \in X.\ \delta'(x) \notin O_d \\ & \quad \text{or } \{T \in \mathsf{TSym}_d \mid \exists x \in X.\ \delta'(x) = T\} = O_d \\ \{T \in \mathsf{TSym}_d \mid \exists x \in X.\ \delta'(x) = T\} & \text{, otherwise} \end{cases} \\
\gamma^{Object}_{O_d,\mathscr{T}'}(X) &= \begin{cases} \{o \in D' \mid \delta'(o) \sqsubseteq' Object\} & \text{, if } X = \top \\ \{o \in D' \mid \bigvee_{T \in X} \delta'(o) = T\} & \text{, otherwise} \end{cases}
\end{aligned}
$$

Figure 3.10: Abstraction and Concretization Functions between $\mathscr{A}^{Object}_{O_d}$ and Concrete Objects in $\mathscr{T}'$

Example 1. *We can consider a simplified Java program containing only the types declared in Listing 3.1. Based on this we have the set of concrete object types $\{Object, B, Null\}$ and the abstract domain $\mathscr{A}^{Object}_{\{Object,B,Null\}}$ as shown in Figure 3.11.*

It is important to point out that although $\mathscr{A}^{Object}_{O_d}$ always has abstract elements $\{Null\}$ and $O_d \setminus \{Null\}$, it is not inherently stronger than the abstract domain $\mathscr{A}^{Object}_{\texttt{null}}$. This is because given a type hierarchy extension $\mathscr{T}'$ which introduces a new dynamic type $d' \sqsubset Object$, for which it holds for some o that $\delta'(o) = d'$, then $\alpha^{Object}_{\texttt{null}}(\{o\}) = \texttt{not-null}$ but $\alpha^{Object}_{O_d,\mathscr{T}'}(\{o\}) = \top$.

3.2.4 Combining the Object Abstract Domains Into One

Of course, we would like just one abstract domain for objects encompassing all of the abstractions discussed in the previous subsections. The abstract domain $\mathscr{A}^{Object}$ for a given type hierarchy $\mathscr{T}$ is a partial Cartesian product of the abstract domains $\mathscr{A}^{Object}_{\texttt{null}}$, $\mathscr{A}^{Object}_{\texttt{length}}$ and $\mathscr{A}^{Object}_{O_d}$ such that the abstraction and concretization functions for any type hierarchy extension $\mathscr{T}'$ can be given as in Figure 3.12. The reason why only a subset of the cartesian product is required is due to the following: As it must hold that $\alpha(\gamma(a)) = a$ for all

```
class Object {...}
abstract class A {...}
interface I {...}
class B extends A implements I {...}
```

Listing 3.1: Type Declarations

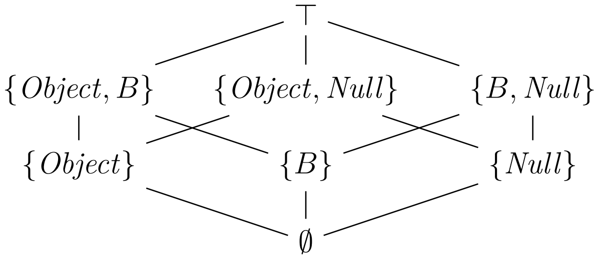


Figure 3.11: Abstract Domain $\mathscr{A}^{\mathit{Object}}_{\{\mathit{Object},B,\mathit{Null}\}}$

$$\alpha^{\mathit{Object}}(X) = (\alpha^{\mathit{Object}}_{\mathtt{null}}(X), \alpha^{\mathit{Object}}_{\mathtt{length}}(X), \alpha^{\mathit{Object}}_{O_d,\mathscr{T}'}(X))$$
$$\gamma^{\mathit{Object}}((a,b,c)) = \gamma^{\mathit{Object}}_{\mathtt{null}}(a) \cap \gamma^{\mathit{Object}}_{\mathtt{length}}(b) \cap \gamma^{\mathit{Object}}_{O_d,\mathscr{T}'}(c)$$

Figure 3.12: Abstraction and Concretization Functions between $\mathscr{A}^{\mathit{Object}}$ and Concrete Objects in $\mathscr{T}'$

abstract elements a, there can never be more than one abstract element representing the same set. We therefore cannot have both $(\bot, y, z)$ and $(x, \bot, z)$ as separate abstract elements, as intuitively both of these would have to represent the empty set. Additionally, while the abstraction for `length` is orthogonal to the abstractions for `null` and exact type (due to the function *length* being defined for all objects, including `null` and non-array types), the abstractions for `null` and exact type are not. While it is true that in one abstraction we may know that `null` does not appear, while in the other abstraction we do not, it is nonetheless impossible for certain abstract elements to be combined without representing the empty set, for example the abstract elements `null` and $\{\mathit{Object}\}$.

The abstract domain $\mathscr{A}^{\mathit{Object}}$ is shown in Figure 3.13.

$$\mathscr{A}^{\mathit{Object}} = (A^{\mathit{Object}}, \sqsubseteq^{\mathit{Object}}, \sqcup^{\mathit{Object}})$$
$$A^{\mathit{Object}} \subset A^{\mathit{Object}}_{\mathtt{null}} \times A^{\mathit{Object}}_{\mathtt{length}} \times A^{\mathit{Object}}_{O_d}$$
$$(a,b,c) \sqsubseteq^{\mathit{Object}} (x,y,z) = a \sqsubseteq^{\mathit{Object}}_{\mathtt{null}} x \land b \sqsubseteq^{\mathit{Object}}_{\mathtt{length}} y \land c \sqsubseteq^{\mathit{Object}}_{O_d} z$$
$$(a,b,c) \sqcup^{\mathit{Object}} (x,y,z) = (a \sqcup^{\mathit{Object}}_{\mathtt{null}} x, b \sqcup^{\mathit{Object}}_{\mathtt{length}} y, c \sqcup^{\mathit{Object}}_{O_d} z)$$

Figure 3.13: Abstract Domain $\mathscr{A}^{\mathit{Object}}$

4 Abstract Domain for Heaps

Joining values which differ between two updates must also include the ability to join heaps, as often updates will also contain a new value for the state of the program heap.

Example 2 (Updates to `heap`)**.** *Symbolically executing the sequent*

$$o \neq \mathit{null}, x \geq 0, x < \mathit{length}(o) \Longrightarrow \{\mathtt{a} := o \parallel \mathtt{i} := x \parallel \mathtt{heap} := h\}[\mathtt{a[i]\ =\ 5;}]\phi$$

will result in the sequent

$$o \neq \mathit{null}, x \geq 0, x < \mathit{length}(o) \Longrightarrow \{\mathtt{a} := o \parallel \mathtt{i} := x \parallel \mathtt{heap} := \mathit{store}(h, o, \mathit{arr}(x), 5)\}\phi$$

containing a modification to the update of the program variable `heap` *of type Heap.*

The simplest and least useful abstraction for heaps would be to have the minimal abstract domain:

$$\begin{array}{c}\top\\ |\\ \bot\end{array}$$

This would imply that at any point when joining heaps, if these are not identical, i.e. if there have been any changes to the program heap whatsoever, that all knowledge of the program heap be forgotten, replaced with a fresh heap for which we do not know anything.

We would, of course, much rather have more refined options available for abstracting perhaps only parts of the heap, or at the very least retaining some information of the shape of structures, or existence of objects that must reside on the heap.

4.1 Normal-Form Heap Abstraction

In general, we are interested only in heaps in D^{Heap} for terms in Trm_{Heap} which can be generated by our program rules, as for the generation of specifications for Java programs we need only consider heaps which can occur while executing said Java program. Initially, any program analysis can be said to begin with the program variable `heap` set to a heap term *heap*, for which the *wellFormed* predicate holds. Well-formedness of heaps entails in particular that only a finite number of objects on the heap are created, i.e. have their *created*-flag set. Further, the program rules can only modify this heap term through the use of *store*, *create* and *anon*, while the heap simplification rules can only reorder, remove or replace these modifications (replacing them with others from the same set). We can therefore define the set of normal-form syntactic representations of heaps as the smallest subset of all syntactic heaps ($\mathit{Heap}^{NF} \subset \mathsf{Trm}_{Heap}$) fulfilling the following properties:

$$\begin{aligned}
\mathit{heap} &\in \mathit{Heap}^{NF}\\
\mathit{create}(h, o) &\in \mathit{Heap}^{NF}, \text{ for all } h \in \mathit{Heap}^{NF}, o : \mathit{Object}\\
\mathit{store}(h, o, f, v) &\in \mathit{Heap}^{NF}, \text{ for all } h \in \mathit{Heap}^{NF}, o : \mathit{Object}, f : \mathit{Field}, v : \mathit{Any}\\
\mathit{anon}(h, \mathit{ls}, h') &\in \mathit{Heap}^{NF}, \text{ for all } h \in \mathit{Heap}^{NF}, o : \mathit{Object}, \mathit{ls} : \mathit{LocSet}, h' : \mathit{Heap}, \text{ if } \mathit{wellFormed}(h') \text{ holds}
\end{aligned}$$

The semantic heap functions represented by elements of Heap^{NF} are well-formed, as was shown in [63]. Reducing our concrete domain to normal-form heaps allows for a better abstract domain, as well as better manipulation at the syntax level, allowing more flexibility when generating specifications. We will show that extending this to the full domain of heaps is then rather trivial.

In particular, joining heaps will only ever occur between heaps, or rather their abstractions, which are both based on the same initial heap. This is ensured by all normal-form heaps being based on the inital value *heap*. But often a larger heap can be chosen, such as the heap before execution of a loop or method call, so we define the family of normal-form heaps Heap^{NF}_{old} to be all normal form heaps based on the normal form heap *old*. In particular, therefore, $\mathit{Heap}^{NF}_{heap} = \mathit{Heap}^{NF}$.

In order to define a concrete domain of normal form heaps for the term *old*, we must fix the semantic value of *old* to a heap $h \in D^{Heap}$. We therefore introduce the family of concrete domains for Heap^{NF}_{old} where *old* is fixed to the value h as:

$$D_h^{\mathit{Heap}^{NF}_{old}} = \{h' \in D^{Heap} \mid \mathit{wellFormed}^{\mathscr{M}}(h') \wedge (h(o, \mathit{created}^{\mathscr{M}}) = tt \rightarrow h'(o, \mathit{created}^{\mathscr{M}}) = tt)\}$$

We define $LS \subset D^{LocSet}$ to contain all object/field pairs not containing the *created* field, i.e.:

$$LS = D^{Object} \times (D^{Field} \setminus \{\mathit{created}^{\mathscr{M}}\})$$

The abstract domain $\mathscr{A}^{LS} = (A^{LS}, \sqsubseteq^{LS}, \sqcup^{LS})$ is defined as:

$$\begin{aligned}
A^{LS} &= \{\bot\} \cup 2^{LS} \\
x \sqcup^{LS} y &= \begin{cases} x & \text{, if } y = \bot \\ y & \text{, if } x = \bot \\ x \cup y & \text{, otherwise} \end{cases} \\
x \sqsubseteq^{LS} y &= \begin{cases} tt & \text{, if } x = \bot \\ \mathit{ff} & \text{, if } x \neq \bot \text{ and } y = \bot \\ x \subseteq y & \text{, otherwise} \end{cases}
\end{aligned}$$

Lemma 1. $\sqcup^{LS}$ *is commutative*

Proof.

$$\begin{aligned}
x \sqcup^{LS} y &= \begin{cases} x & \text{, if } y = \bot \\ y & \text{, if } x = \bot \\ x \cup y & \text{, otherwise} \end{cases} \\
&= \begin{cases} x & \text{, if } y = \bot \\ y & \text{, if } x = \bot \\ y \cup x & \text{, otherwise} \end{cases} \\
&= y \sqcup^{LS} x
\end{aligned}$$

□

When joining two abstract heaps based on some initial heap *old* we can simply use the union of both sets to gain all possible heaps in either abstraction. However, due to the abstract domain containing infinite ascending chains, we could not simply use joining to reach a fixed point for our abstract heaps and therefore require the use of a widening operator.

We must choose our widening operator carefully to ensure that the result remains fairly precise whenever possible. We can, however, use knowledge of how Java programs work in order to know where precision matters most. An example of this is *object field* modifications (and also *static field* modifications, as these are modelled as object field modifications of the object *null*): ignoring arrays for the moment, any given Java program defines by its source code a finite set of object fields at which the program can at most modify objects. Simply put, the *null* object can only be modified at most at static fields, while a non-*null*, non-array object can only be modified at most at instance fields declared in the program, as well as at the *created* field. Any given program can only declare a finite number of static and instance fields. We can further constrain this finite set using static analysis to over-approximate the set of fields at which objects are actually modified at by examining the left hand sides of all assignments appearing in the program. If a field is never used in an assignment, no objects will be modified at that field by the given program. KeY performs a similar analysis when determining which local variables might be modified within a loop. Here, however, we focus not on *variables* on the left hand side of assignments, but rather the *fields*. Knowing this, we can define our widening based on a finite set of non-array fields *fs*, thereby narrowing one otherwise infinite dimension.

Getting back to arrays, we cannot give a finite set of array fields a priori, as while the set of array fields modified in a terminating program will of course be finite, it cannot be determined by mere static code analysis, as the array indices are expressions which must be evaluated, rather than mere field names which are static. Our widening must therefore be able to deal with this possibly infinite dimension.

Let the set of all array index accessor fields be defined as

$$Arr = \{arr^{\mathcal{M}}(x) \mid x \in \mathbb{Z}\} \subset (D^{Field} \setminus \{created^{\mathcal{M}}\})$$

Lemma 2. $|Arr| = \infty$

Proof. As $arr^{\mathcal{M}}$ is injective, it follows that $|Arr| = |\{arr^{\mathcal{M}}(x) \mid x \in \mathbb{Z}\}| = \infty$ □

The third possibly infinite dimension is objects modified. Once again, the problem is an inability to provide a finite set of objects modified a priori, again due to object references being expressions needing evaluation.

In short, our widening operator must be able to deal with the following infinite ascending chains for all $i, j \in \mathbb{N}$, $o, o_i, o_j \in D^{Object}$, $f, f_i, f_j \in (D^{Field} \setminus Arr)$ with $o_i = o_j \to i = j$ and $f_i = f_j \to i = j$, in particular:

$$\begin{aligned}
&\langle a_z \rangle && \text{, with } a_0 = \emptyset, a_{i+1} = a_i \cup \{(o_i, f)\} \\
&\langle b_z \rangle && \text{, with } b_0 = \emptyset, b_{i+1} = b_i \cup \{(o, arr^{\mathcal{M}}(i))\} \\
&\langle c_z \rangle && \text{, with } c_0 = \emptyset, c_{i+1} = c_i \cup \{(o_i, arr^{\mathcal{M}}(i))\} \\
&\langle d_z \rangle && \text{, with } d_0 = \emptyset, d_{i+1} = d_i \cup \{(o_i, f_i)\}
\end{aligned}$$

We define a family of widening operators $\nabla_{fs,n,m,k}$, with $fs \subset ((D^{Field} \setminus Arr) \setminus \{created^{\mathcal{M}}\})$, $|fs| < \infty$, $n, m, k \in \mathbb{N}$ as follows:

$$\begin{aligned}
\nabla_{fs,n,m,k} &: A^{LS} \times A^{LS} \to A^{LS} \\
&(x, y) \mapsto \begin{cases} \bot & \text{, if } x \sqcup^{LS} y = \bot \\ W_{fs,n,m,k}(x \sqcup^{LS} y) & \text{, otherwise} \end{cases}
\end{aligned}$$

Where $W_{fs,n,m,k}$ is defined as:

$$W_{fs,n,m,k} : 2^{LS} \to 2^{LS}$$
$$ls \mapsto \begin{cases} LS & \text{, if } \exists (o', f') \in ls.\ f' \notin (fs \cup Arr) \\ ls \cup W^N_{fs,n}(ls) \cup W^M_m(ls) \cup W^K_k(ls) & \text{, otherwise} \end{cases}$$

With $W^N_{fs,n}, W^M_m, W^K_k$ defined as:

$$\begin{aligned} W^N_{fs,n}(ls) &= \{(o,f) \mid f \in fs \land |\{o' \mid (o', f) \in ls\}| > n\} \\ W^M_m(ls) &= \{(o,f) \mid f \in Arr \land |\{f' \in Arr \mid (o, f') \in ls\}| > m\} \\ W^K_k(ls) &= \begin{cases} D^{Object} \times Arr & \text{, if } |\{o \mid \exists f \in Arr.\ (o,f) \in ls\}| > k \\ \emptyset & \text{, otherwise} \end{cases} \end{aligned}$$

Intuitively, $W_{fs,n,m,k}(ls)$ widens the location set ls in the four possible infinite dimensions demonstrated in the sequences $\langle a_z \rangle, \dots, \langle d_z \rangle$ based on the bounds fs, n, m, k as follows:

1. The set fs bounds the number of different object fields within ls, widening in all dimensions if an object field outside of fs is present. This widening would be applied to sequence $\langle d_z \rangle$.
2. $W^N_{fs,n}$ bounds the number of different objects with the same object field in ls, widening to include all objects for that object field if the bound n is surpassed. This widening would be applied to sequence $\langle a_z \rangle$.
3. $W^M_m(ls)$ bounds the number of different array index fields for the same object within ls, widening to include all array index fields for that object if the bound m is passed. This widening would be applied to sequence $\langle b_z \rangle$.
4. Finally, $W^K_k(ls)$ bounds the number of different objects paired with an array index field in ls, widening to include all pairs of objects and array index fields if the bound k is passed. This widens both the infinite dimension of a single object being paired with an infinite number of array index fields, as well as the diagonal of infinite objects with infinite array index fields. This widening would be applied to sequence $\langle c_z \rangle$.

Lemma 3. $\hat{f} \notin Arr \to (\hat{o}, \hat{f}) \notin W^M_m(X) \land (\hat{o}, \hat{f}) \notin W^K_k(X)$

Proof. This is trivial, as by definition

$$\begin{aligned} (\hat{o}, \hat{f}) \in W^M_m(X) &\equiv (\hat{o}, \hat{f}) \in \{(o,f) \mid f \in Arr \land |\{f' \mid f' \in Arr \land (o, f') \in X\}| > m\} \\ &\equiv \hat{f} \in Arr \land |\{f' \mid f' \in Arr \land (\hat{o}, f') \in X\}| > m \\ &\Rightarrow \hat{f} \in Arr \end{aligned}$$

$$\begin{aligned} (\hat{o}, \hat{f}) \in W^K_k(X) &\equiv (\hat{o}, \hat{f}) \in \begin{cases} D^{Object} \times Arr & \text{, if } |\{o' \mid f' \in Arr \land (o', f') \in X\}| > k \\ \emptyset & \text{, otherwise} \end{cases} \\ &\equiv \begin{cases} (\hat{o}, \hat{f}) \in (D^{Object} \times Arr) & \text{, if } |\{o' \mid f' \in Arr \land (o', f') \in X\}| > k \\ (\hat{o}, \hat{f}) \in \emptyset & \text{, otherwise} \end{cases} \\ &\equiv \begin{cases} (\hat{o}, \hat{f}) \in (D^{Object} \times Arr) & \text{, if } |\{o' \mid f' \in Arr \land (o', f') \in X\}| > k \\ ff & \text{, otherwise} \end{cases} \\ &\Rightarrow (\hat{o}, \hat{f}) \in (D^{Object} \times Arr) \\ &\Rightarrow \hat{f} \in Arr \end{aligned}$$

□

Example 3 (Widening Heaps). *We show that the infinite ascending chain $\langle a_z \rangle$ as above, with $a_0 = \emptyset$ and $a_{i+1} = a_i \cup \{(o_i, f)\}$, reaches a fixed point through the widening operator $\nabla_{fs,n,m,k}$, i.e. that the sequence $\langle x'_z \rangle$ (defined by $x'_0 = \bot$ and $x'_{i+1} = x'_i \nabla_{fs,n,m,k} a_i$) is ultimately stationary. We know that for all $i, j \in \mathbb{N}$, $o_i, o_j \in D^{Object}$, with $o_i = o_j \rightarrow i = j$:*

$$\begin{aligned}
x'_1 &= (x'_0 \ \nabla_{fs,n,m,k} \ a_0) = (\bot \ \nabla_{fs,n,m,k} \ \emptyset) = W_{fs,n,m,k}(\emptyset) = \emptyset \\
x'_2 &= (x'_1 \ \nabla_{fs,n,m,k} \ a_1) = (\emptyset \ \nabla_{fs,n,m,k} \ \{(o_1, f)\}) = W_{fs,n,m,k}(\{(o_1, f)\}) \\
x'_{i+3} &= (x'_i \ \nabla_{fs,n,m,k} \ a_i) = W_{fs,n,m,k}(x'_i \cup a_i)
\end{aligned}$$

We consider two cases:

$f \notin fs$:

$$\begin{aligned}
x'_2 &= W_{fs,n,m,k}(\{(o_1, f)\}) = LS \\
x'_{i+3} &= W_{fs,n,m,k}(x'_{i+2} \cup a_{i+2}) = W_{fs,n,m,k}(LS \cup a_{i+2}) = W_{fs,n,m,k}(LS) = LS
\end{aligned}$$

So widening finds the fixed point LS.

$f \in fs$: *We can show that*

$$\forall i \in \mathbb{N}.\ i < n \rightarrow x'_{i+2} = a_{i+1} \tag{4.1}$$

Proof. By induction over i:

Base case: For $i = 0 < n$ it holds that

$$\begin{aligned}
x'_{i+2} &= x'_2 \\
&= W_{fs,n,m,k}(\{(o_1, f)\}) \\
&= \{(o_1, f)\} \cup W^N_{fs,n}(\{(o_1, f)\}) \cup W^M_m(\{(o_1, f)\}) \cup W^K_k(\{(o_1, f)\}) \\
&= \{(o_1, f)\} \\
&= a_1 \\
&= a_{i+1}
\end{aligned}$$

Step case: For $i > 0$ and $i < n$ we can assume the induction hypothesis:

$$x'_{i+1} = a_i \tag{IH1}$$

Then it holds that

$$\begin{aligned}
x'_{i+2} &= W_{fs,n,m,k}(x'_{i+1} \cup a_{i+1}) \\
&= W_{fs,n,m,k}(a_i \cup a_{i+1}) && \text{by (IH1)} \\
&= W_{fs,n,m,k}(a_i \cup a_i \cup \{(o_i, f)\}) && \text{by definition} \\
&= W_{fs,n,m,k}(a_i \cup \{(o_i, f)\}) \\
&= W_{fs,n,m,k}(a_{i+1}) \\
&= a_{i+1} \cup W^N_{fs,n}(a_{i+1}) \cup W^M_m(a_{i+1}) \cup W^K_k(a_{i+1}) \\
&= a_{i+1} \cup W^N_{fs,n}(a_{i+1}) && \text{by Lemma 3, as } a_{i+1} \text{ contains only pairs with field } f \notin Arr \\
&= a_{i+1} && \text{as } a_{i+1} \text{ contains only } i+1 \text{ pairs } (o', f) \text{ and } i+1 \leq n
\end{aligned}$$

And thus we have shown (4.1). □

Using (4.1) *we can show that*

$$\forall i \in \mathbb{N}.\ i \geq n \to x'_{i+2} = \{(o', f) \mid o' \in D^{\mathit{Object}}\} \tag{4.2}$$

Proof. By induction over i:

Base case: For $i = 0 \geq n$ it holds that $n = 0$ and

$$\begin{aligned}
x'_{i+2} &= x'_2 \\
&= W_{\mathit{fs},n,m,k}(\{(o_1, f)\}) \\
&= \{(o_1, f)\} \cup W^N_{\mathit{fs},n}(\{(o_1, f)\}) \cup \emptyset \cup \emptyset && \text{by Lemma (3)} \\
&= \{(o_1, f)\} \cup \{(o, f') \mid o \in D^{\mathit{Object}} \wedge f' \in \mathit{fs} \wedge |\{o' \mid (o', f') \in \{(o_1, f)\}\}| > n\} \\
&= \{(o_1, f)\} \cup \{(o, f) \mid o \in D^{\mathit{Object}} \wedge |\{o' \mid (o', f) \in \{(o_1, f)\}\}| > n\} \\
& && \text{as no field other than } f \text{ can occur more than } n = 0 \text{ times} \\
&= \{(o_1, f)\} \cup \{(o, f) \mid o \in D^{\mathit{Object}} \wedge 1 > n\} \\
&= \{(o_1, f)\} \cup \{(o, f) \mid o \in D^{\mathit{Object}}\} \\
&= \{(o', f) \mid o' \in D^{\mathit{Object}}\}
\end{aligned}$$

Step case: For $i > 0$ and $i > n$ $(i - 1 \geq n)$ we can assume the induction hypothesis:

$$x'_{i+1} = \{(o', f) \mid o' \in D^{\mathit{Object}}\} \tag{IH2}$$

By case distinction

either $i = n$:

$$\begin{aligned}
x'_{i+2} &= W_{\mathit{fs},n,m,k}(x'_{i+1} \cup a_{i+1}) \\
&= W_{\mathit{fs},n,m,k}(x'_{n+1} \cup a_{n+1}) \\
&= W_{\mathit{fs},n,m,k}(x'_{(n-1)+2} \cup a_{n+1}) \\
&= W_{\mathit{fs},n,m,k}(a_{(n-1)+1} \cup a_{n+1}) && \text{by (4.1)} \\
&= W_{\mathit{fs},n,m,k}(a_n \cup a_{n+1}) \\
&= W_{\mathit{fs},n,m,k}(a_{n+1}) \\
&= a_{n+1} \cup W^N_{\mathit{fs},n}(a_{n+1}) \cup \emptyset \cup \emptyset \\
& && \text{by Lemma (3), as } a_{n+1} \text{ contains only pairs with field } f \notin \mathit{Arr} \\
&= a_{n+1} \cup \{(o, f') \mid f' \in \mathit{fs} \wedge |\{o' \mid (o', f') \in \{(o_1, f)\}\}| > n\} \\
&= a_{n+1} \cup \{(o, f) \mid o \in D^{\mathit{Object}}\} \\
& && \text{as } a_{n+1} \text{ contains exactly } n+1 \text{ pairs } (o', f) \text{ and } n+1 > n \\
&= \{(o', f) \mid o' \in D^{\mathit{Object}}\}
\end{aligned}$$

or $i > n$:

$$\begin{aligned}
x'_{i+2} &= W_{\mathit{fs},n,m,k}(x'_{i+1} \cup a_{i+1}) \\
&= W_{\mathit{fs},n,m,k}(\{(o', f) \mid o' \in D^{\mathit{Object}}\} \cup a_{i+1}) && \text{by (IH2)} \\
&= W_{\mathit{fs},n,m,k}(\{(o', f) \mid o' \in D^{\mathit{Object}}\}) \\
&= \{(o', f) \mid o' \in D^{\mathit{Object}}\} \cup W^N_{\mathit{fs},n}(\{(o', f) \mid o' \in D^{\mathit{Object}}\}) \cup \emptyset \cup \emptyset \\
& && \text{by Lemma (3), as } f \notin \mathit{Arr} \\
&= \{(o', f) \mid o' \in D^{\mathit{Object}}\} \cup \{(o', f) \mid o' \in D^{\mathit{Object}}\} \\
&= \{(o', f) \mid o' \in D^{\mathit{Object}}\}
\end{aligned}$$

And thus we have shown (4.2). □

So the fixed point $\{(o', f) \mid o' \in D^{\mathit{Object}}\}$ *will be reached by widening* $\langle a_i \rangle$.

We will now prove that $\nabla_{fs,n,m,k}$ is a widening operator according to Definition 9. For this proof we require the introduction of various helper functions and many lemmas about the helper functions, $W_{fs,n,m,k}$ and $\nabla_{fs,n,m,k}$. This is structured as follows:

- In Subsection 4.2.1 we introduce helper functions and lemmas related to the widening aspect on object fields.
- In Subsection 4.2.2 we introduce helper functions and lemmas related to the widening aspect on array index fields.
- Subsection 4.2.3 introduces one last helper function in order to combine the two widening aspects and finally proves that $\nabla_{fs,n,m,k}$ is a widening operator.

The lemmas in the following sections contain detailed proofs. As to the best of our knowledge this is the first non-trivial widening operator defined for an abstraction of heaps which tries to retain full precision of which locations on the heap have been modified if these locations are not responsible for the widening, the mathematically rigorous proofs ensure that this is indeed a sound widening operator.

Note: free variables in lemmas are to be considered universally quantified.

4.2.1 Widening Object Fields

We introduce the following helper functions:

$$\mathit{finFieldMods}_{fs}(\mathit{ls}) = \{f \mid f \in \mathit{fs} \wedge |\{o' \mid (o', f) \in \mathit{ls}\}| < \infty\}$$

$$\mathit{fieldModsLeft}_{fs,n}(\mathit{ls}) = \sum_{f \in \mathit{finFieldMods}_{fs}(\mathit{ls})} 1 + n - |\{o' \mid (o', f) \in \mathit{ls}\}|$$

The function $\mathit{finFieldMods}_{fs}$ looks at a location set and returns the subset of fs for which only a finite number of *field modifications* exist, i.e. pairs of objects and fields in fs. Intuitively, $\mathit{fieldModsLeft}_{fs,n}$ gives an upper bound of the number of widening steps involving new field modifications left until no more widening on these dimensions is possible. In essence $\mathit{fieldModsLeft}_{fs,n}(X)$ computes an upper bound for the maximal number of iterations the following algorithm can have:

1. Set X to $W_{fs,n,m,k}(X \cup (o, f))$, where $(o, f) \notin X$ and f is not an array index field.
2. If $X \neq LS$, goto 1.

Lemma 4. $X \subseteq Y \rightarrow \mathit{finFieldMods}_{fs}(Y) \subseteq \mathit{finFieldMods}_{fs}(X)$

Proof. This follows directly from the definition of $\mathit{finFieldMods}_{fs}$:

$$\begin{aligned} f \in \mathit{finFieldMods}_{fs}(Y) &\equiv f \in \mathit{fs} \wedge |\{o' \mid (o', f) \in Y\}| < \infty \\ &\Rightarrow f \in \mathit{fs} \wedge |\{o' \mid (o', f) \in X\}| < \infty && \text{as } X \subseteq Y \\ &\equiv f \in \mathit{finFieldMods}_{fs}(X) && \square \end{aligned}$$

Lemma 5. $\hat{f} \in \mathit{fs} \rightarrow \{\hat{o} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} = D^{\mathit{Object}} \vee |\{\hat{o} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\}| \leq n$

Proof. As $\hat{f} \in \mathit{fs}$ and $\mathit{fs} \subset ((D^{\mathit{Field}} \setminus \mathit{Arr}) \setminus \{\mathit{created}^{\mathscr{M}}\})$, we know

$$\hat{f} \neq \mathit{created}^{\mathscr{M}} \tag{4.3}$$

$$\hat{f} \notin \mathit{Arr} \tag{4.4}$$

By case distinction:

$\exists (o', f') \in X.\ f' \notin (\mathit{fs} \cup \mathit{Arr})$:

$$\begin{aligned} \{\hat{o} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} &= \{\hat{o} \mid (\hat{o}, \hat{f}) \in \mathit{LS}\} \\ &= \{\hat{o} \mid (\hat{o}, \hat{f}) \in (D^{\mathit{Object}} \times (D^{\mathit{Field}} \setminus \{\mathit{created}^{\mathscr{M}}\})\} \\ &= D^{\mathit{Object}} && \text{by (4.3)} \end{aligned}$$

$\forall (o', f') \in X.\ f' \in (\mathit{fs} \cup \mathit{Arr})$:

$$\begin{aligned} &\{\hat{o} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} \\ =\ &\{\hat{o} \mid (\hat{o}, \hat{f}) \in (X \cup W^N_{\mathit{fs},n}(X) \cup W^M_m(X) \cup W^K_k(X))\} \\ =\ &\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^N_{\mathit{fs},n}(X) \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in W^K_k(X)\} \\ =\ &\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^N_{\mathit{fs},n}(X)\} && \text{by (4.4) and Lemma 3} \\ =\ &\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, f) \mid f \in \mathit{fs} \wedge |\{o' \mid (o', f) \in X\}| > n\}\} \\ =\ &\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, \hat{f}) \mid |\{o' \mid (o', \hat{f}) \in X\}| > n\}\} \end{aligned}$$

We further distinguish between two cases:

$|\{o' \mid (o', \hat{f}) \in X\}| > n$:

$$\begin{aligned} \{\hat{o} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} &= \{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, \hat{f}) \mid |\{o' \mid (o', \hat{f}) \in X\}| > n\}\} \\ &= \{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, \hat{f}) \mid \mathit{tt}\}\} \\ &= \{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee \mathit{tt}\} \\ &= \{\hat{o} \mid \mathit{tt}\} \\ &= D^{\mathit{Object}} \end{aligned}$$

$|\{o' \mid (o', \hat{f}) \in X\}| \leq n$:

$$\begin{aligned} |\{\hat{o} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\}| &= |\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, \hat{f}) \mid |\{o' \mid (o', \hat{f}) \in X\}| > n\}\}| \\ &= |\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, \hat{f}) \mid \mathit{ff}\}\}| \\ &= |\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \emptyset\}| \\ &= |\{\hat{o} \mid (\hat{o}, \hat{f}) \in X \vee \mathit{ff}\}| \\ &= |\{\hat{o} \mid (\hat{o}, \hat{f}) \in X\}| \\ &= |\{o' \mid (o', \hat{f}) \in X\}| \\ &\leq n \end{aligned}$$

□

Lemma 6. $f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) \to 0 < 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| \leq n + 1$

Proof.

$$\begin{aligned}
f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) &\equiv f \in fs \land |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| < \infty \\
&\Rightarrow |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| < \infty \\
&\Rightarrow \{o' \mid (o', f) \in W_{fs,n,m,k}(X)\} \neq D^{\mathit{Object}} \\
&\Rightarrow |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| \leq n && \text{by Lemma 5} \\
&\Rightarrow 0 < 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| \leq n + 1 && \square
\end{aligned}$$

Lemma 7. $0 \leq \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) \leq |fs|(n+1)$

Proof. By case distinction:

$\exists (o', f') \in X.\ f' \notin (fs \cup \mathit{Arr})$:

$$\mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) = \mathit{fieldModsLeft}_{fs,n}(\mathit{LS}) = 0$$

$\forall (o', f') \in X.\ f' \in (fs \cup \mathit{Arr})$:

$$\mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) = \sum_{\hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(X)\}|$$

By definition

$$\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) = \{f \mid f \in fs \land |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(X)\}| < \infty\} \subseteq fs \tag{4.5}$$

We can determine a lower bound:

$$\begin{aligned}
\mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) &= \sum_{\hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(X)\}| \\
&\geq \sum_{\hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 && \text{by Lemma 6} \\
&\geq 0
\end{aligned}$$

As well as determining an upper bound:

$$\begin{aligned}
\mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) &= \sum_{\hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(X)\}| \\
&\leq \sum_{\hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} n + 1 && \text{by Lemma 6} \\
&\leq \sum_{\hat{f} \in fs} n + 1 && \text{by (4.5)} \\
&= |fs|(n+1) && \square
\end{aligned}$$

Lemma 8.

$$W_{fs,n,m,k}(X) \subseteq W_{fs,n,m,k}(Y) \rightarrow \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) \geq \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(Y))$$

Proof. We know that for any f

$$\begin{aligned} W_{fs,n,m,k}(X) \subseteq W_{fs,n,m,k}(Y) &\Rightarrow \{o' \mid (o',f) \in W_{fs,n,m,k}(X)\} \subseteq \{o' \mid (o',f) \in W_{fs,n,m,k}(Y)\} \\ &\Rightarrow |\{o' \mid (o',f) \in W_{fs,n,m,k}(X)\}| \leq |\{o' \mid (o',f) \in W_{fs,n,m,k}(Y)\}| \end{aligned} \tag{4.6}$$

We can therefore show:

$$\begin{aligned} & \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) \\ = & \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 + n - |\{o' \mid (o',f) \in W_{fs,n,m,k}(X)\}| \\ \geq & \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))} 1 + n - |\{o' \mid (o',f) \in W_{fs,n,m,k}(X)\}| && \text{by Lemmas 4 and 6} \\ \geq & \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))} 1 + n - |\{o' \mid (o',f) \in W_{fs,n,m,k}(Y)\}| && \text{by (4.6)} \\ = & \ \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(Y)) \end{aligned}$$

□

Lemma 9. $\hat{f} \in fs \wedge (\hat{o},\hat{f}) \notin W_{fs,n,m,k}(X) \wedge W_{fs,n,m,k}(X) \subset W_{fs,n,m,k}(Y) \wedge (\hat{o},\hat{f}) \in W_{fs,n,m,k}(Y) \rightarrow$
$\mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) > \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(Y))$

Proof. We know that for any f

$$\begin{aligned} W_{fs,n,m,k}(X) \subset W_{fs,n,m,k}(Y) &\Rightarrow \{o' \mid (o',f) \in W_{fs,n,m,k}(X)\} \subseteq \{o' \mid (o',f) \in W_{fs,n,m,k}(Y)\} \\ &\Rightarrow |\{o' \mid (o',f) \in W_{fs,n,m,k}(X)\}| \leq |\{o' \mid (o',f) \in W_{fs,n,m,k}(Y)\}| \end{aligned} \tag{4.7}$$

Furthermore,

$$\begin{aligned} (\hat{o},\hat{f}) \notin W_{fs,n,m,k}(X) &\Rightarrow \hat{o} \notin \{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\} \\ &\Rightarrow \{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\} \neq D^{Object} \\ &\Rightarrow |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\}| \leq n && \text{by Lemma 5} \\ &\Rightarrow \hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) && (4.8) \end{aligned}$$

By case distinction

$\hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))$: Then we have

$$\begin{aligned} \hat{f} \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y)) &\equiv \hat{f} \in fs \wedge |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(Y)\}| < \infty \\ &\Rightarrow |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(Y)\}| < \infty \end{aligned} \tag{4.9}$$

As well as

$$\begin{aligned} & W_{fs,n,m,k}(X) \subset W_{fs,n,m,k}(Y) \\ \Rightarrow & \ W_{fs,n,m,k}(X) \subseteq (W_{fs,n,m,k}(Y) \setminus \{(\hat{o},\hat{f})\}) && \text{as } (\hat{o},\hat{f}) \notin W_{fs,n,m,k}(X) \\ \Rightarrow & \ \{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\} \subseteq \{o' \mid (o',\hat{f}) \in (W_{fs,n,m,k}(Y) \setminus \{(\hat{o},\hat{f})\})\} \\ \Rightarrow & \ \{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\} \subseteq (\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(Y)\} \setminus \{\hat{o}\}) \\ \Rightarrow & \ |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\}| \leq |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(Y)\} \setminus \{\hat{o}\}| \\ \Rightarrow & \ |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\}| \leq |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(Y)\}| - 1 \\ & && \text{by (4.9) and } (\hat{o},\hat{f}) \in W_{fs,n,m,k}(Y) \\ \Rightarrow & \ |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(X)\}| < |\{o' \mid (o',\hat{f}) \in W_{fs,n,m,k}(Y)\}| && (4.10) \end{aligned}$$

Therefore

$$\begin{aligned}
& \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) \\
&= \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| \\
&= 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(X)\}| \\
&\quad + \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.8)} \\
&> 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(Y)\}| \\
&\quad + \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.10)} \\
&\geq 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(Y)\}| \\
&\quad + \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| && \text{by Lemmas 6 and 4} \\
&\geq 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(Y)\}| \\
&\quad + \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(Y)\}| && \text{by (4.7)} \\
&= \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(Y)\}| \\
&= \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(Y))
\end{aligned}$$

$\hat{f} \notin \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))$:

$$\begin{aligned}
& \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(X)) \\
&= \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X))} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| \\
&= 1 + n - |\{o' \mid (o', \hat{f}) \in W_{fs,n,m,k}(X)\}| \\
&\quad + \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.8)} \\
&> \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(X)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.8) and Lemma 6} \\
&\geq \sum_{f \in (\mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y)) \setminus \{\hat{f}\})} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| && \text{by Lemmas 6 and 4} \\
&\geq \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(X)\}| \\
& && \text{by case distinction assumption} \\
&\geq \sum_{f \in \mathit{finFieldMods}_{fs}(W_{fs,n,m,k}(Y))} 1 + n - |\{o' \mid (o', f) \in W_{fs,n,m,k}(Y)\}| && \text{by (4.7)} \\
&= \mathit{fieldModsLeft}_{fs,n}(W_{fs,n,m,k}(Y)) && \square
\end{aligned}$$

This (with Lemma 8) is the main lemma of this subsection, showing that as more object field modifications exist after a widening, the number of field modifications available decreases, i.e. with Lemma 7 that the number of such steps is bound.

We introduce the following helper functions:

$$\begin{aligned} \mathit{indexMods}(\mathit{ls}) &= \{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in \mathit{ls}\}|\} \\ \mathit{finIndexMods}(\mathit{ls}) &= \{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in \mathit{ls}\}| < \infty\} \\ \mathit{indexModsLeft}_{m,k}(\mathit{ls}) &= (m+1)(k+1-\mathit{min}(k+1, |\ \mathit{indexMods}(\mathit{ls})\ |)) \\ &\quad + \sum_{o \in \mathit{finIndexMods}(\mathit{ls})} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in \mathit{ls}\}| \end{aligned}$$

The function *indexMods* takes a locations set and calculates the set of objects for which at least one *array index modification* exists, i.e. pairs of objects and array index fields. The function *finIndexMods* is the subset of *indexMods* containing only those objects for which a finite number of array index modification exists. Intuitively, the function $\mathit{indexModsLeft}_{m,k}$ calculates the number of widening steps containing such modifications left before the infinite dimensions related to object/index pairs are completely widened. In essence $\mathit{indexModsLeft}_{m,k}(X)$ computes an upper bound on the maximal number of iterations the following algorithm can have:

1. Set X to $W_{\mathit{fs},n,m,k}(X \cup (o, \mathit{arr}^{\mathscr{M}}(i)))$, where $i \in \mathbb{Z}$ and $(o, \mathit{arr}^{\mathscr{M}}(i)) \notin X$.
2. If $X \neq D^{\mathit{Object}} \times \mathit{Arr}$, goto 1.

Lemma 10. $\hat{f} \notin \mathit{fs} \to (\hat{o}, \hat{f}) \notin W^N_{\mathit{fs},n}(X)$

Proof. This is trivial, as by definition

$$\begin{aligned} (\hat{o}, \hat{f}) \in W^N_{\mathit{fs},n}(X) &\equiv (\hat{o}, \hat{f}) \in \{(o,f) \mid f \in \mathit{fs} \land |\{o' \mid (o', f) \in X\}| > n\} \\ &\equiv \hat{f} \in \mathit{fs} \land |\{o' \mid (o', \hat{f}) \in X\}| > n \\ &\Rightarrow \hat{f} \in \mathit{fs} \end{aligned}$$

□

Lemma 11. $\{o \mid \{f \in \mathit{Arr} \mid (o,f) \in W^M_m(X)\} \neq \emptyset\} \subseteq \{o \mid \{f \in \mathit{Arr} \mid (o,f) \in X\} \neq \emptyset\}$

Proof.

$$\begin{aligned} &\hat{o} \in \{o \mid \{f \in \mathit{Arr} \mid (o,f) \in W^M_m(X)\} \neq \emptyset\} \\ \equiv\ &\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W^M_m(X)\} \neq \emptyset \\ \equiv\ &\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in \{(o,f) \mid f \in \mathit{Arr} \land |\{f' \mid f' \in \mathit{Arr} \land (o, f') \in X\}| > m\}\} \neq \emptyset \\ \equiv\ &\{\hat{f} \in \mathit{Arr} \mid \hat{f} \in \mathit{Arr} \land |\{f' \mid f' \in \mathit{Arr} \land (\hat{o}, f') \in X\}| > m\} \neq \emptyset \\ \equiv\ &\{\hat{f} \in \mathit{Arr} \mid\ |\{f' \mid f' \in \mathit{Arr} \land (\hat{o}, f') \in X\}| > m\} \neq \emptyset \\ \equiv\ &\exists f \in \mathit{Arr}.\ f \in \{\hat{f} \in \mathit{Arr} \mid\ |\{f' \mid f' \in \mathit{Arr} \land (\hat{o}, f') \in X\}| > m\} \\ \equiv\ &\exists f \in \mathit{Arr}.\ |\{f' \mid f' \in \mathit{Arr} \land (\hat{o}, f') \in X\}| > m \\ \equiv\ &|\{f' \mid f' \in \mathit{Arr} \land (\hat{o}, f') \in X\}| > m \\ \Rightarrow\ &\{f' \mid f' \in \mathit{Arr} \land (\hat{o}, f') \in X\} \neq \emptyset \\ \equiv\ &\hat{o} \in \{o \mid \{f \in \mathit{Arr} \mid (o,f) \in X\} \neq \emptyset\} \end{aligned}$$

□

Lemma 12. *Similar to Lemma 5 for object fields, widening of array index fields is bound by m or complete:*

$$\hat{o} \in D^{\mathit{Object}} \rightarrow \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} = \mathit{Arr} \vee |\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\}| \leq m$$

Proof. By case distinction:

$\exists (o', f') \in X.\ f' \notin (\mathit{fs} \cup \mathit{Arr})$:

$$\begin{aligned}\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in \mathit{LS}\} \\ &= \{\hat{f} \in \mathit{Arr} \mid \mathit{tt}\} \\ &= \mathit{Arr}\end{aligned}$$

$\forall (o', f') \in X.\ f' \in (\mathit{fs} \cup \mathit{Arr})$:

$$\begin{aligned}&\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in (X \cup W^N_{\mathit{fs},n}(X) \cup W^M_m(X) \cup W^K_k(X))\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^N_{\mathit{fs},n}(X) \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in W^K_k(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in W^K_k(X)\} && \text{by Lemma 10}\end{aligned}$$

A further case distinction gives us

$|\{o \mid \exists f \in \mathit{Arr}.\ (o, f) \in \mathit{ls}\}| > k$:

$$\begin{aligned}&\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in W^K_k(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in (D^{\mathit{Object}} \times \mathit{Arr})\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee \mathit{tt}\} && \text{as } \hat{f} \in \mathit{Arr} \\ &= \{\hat{f} \in \mathit{Arr} \mid \mathit{tt}\} \\ &= \mathit{Arr}\end{aligned}$$

$|\{o \mid \exists f \in \mathit{Arr}.\ (o, f) \in \mathit{ls}\}| \leq k$:

$$\begin{aligned}&\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in W^K_k(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X) \vee (\hat{o}, \hat{f}) \in \emptyset\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^M_m(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in \{(o, f) \mid f \in \mathit{Arr} \wedge |\{f' \mid f' \in \mathit{Arr} \wedge (o, f') \in X\}| > m\}\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee (\hat{f} \in \mathit{Arr} \wedge |\{f' \mid f' \in \mathit{Arr} \wedge (\hat{o}, f') \in X\}| > m)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee |\{f' \mid f' \in \mathit{Arr} \wedge (\hat{o}, f') \in X\}| > m\}\end{aligned}$$

One last case distinction yields

$|\{f' \mid f' \in \mathit{Arr} \wedge (\hat{o}, f') \in X\}| > m$:

$$\begin{aligned}&\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee |\{f' \mid f' \in \mathit{Arr} \wedge (\hat{o}, f') \in X\}| > m\} \\ &= \{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee \mathit{tt}\} \\ &= \{\hat{f} \in \mathit{Arr} \mid \mathit{tt}\} \\ &= \mathit{Arr}\end{aligned}$$

$|\{f' \mid f' \in \mathit{Arr} \wedge (\hat{o}, f') \in X\}| \leq m$:

$$\begin{aligned}
&|\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X)\}| \\
={}& |\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee |\{f' \mid f' \in \mathit{Arr} \wedge (\hat{o}, f') \in X\}| > m\}| \\
={}& |\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X \vee \mathit{ff}\}| \\
={}& |\{\hat{f} \in \mathit{Arr} \mid (\hat{o}, \hat{f}) \in X\}| \\
\leq{}& m
\end{aligned}$$

□

Lemma 13. $X \subseteq Y \rightarrow \mathit{finIndexMods}(Y) \subseteq ((\mathit{indexMods}(Y) \setminus \mathit{indexMods}(X)) \cup \mathit{finIndexMods}(X))$

Proof. We show

$$X \subseteq Y \wedge o \in \mathit{finIndexMods}(Y) \rightarrow o \in ((\mathit{indexMods}(Y) \setminus \mathit{indexMods}(X)) \cup \mathit{finIndexMods}(X))$$

As $X \subseteq Y$, we know that

$$\begin{aligned}
o \in \mathit{finIndexMods}(Y) &\equiv 0 < |\{f \in \mathit{Arr} \mid (o, f) \in Y\}| < \infty \\
&\equiv 0 < |\{f \in \mathit{Arr} \mid (o, f) \in Y\}| \wedge |\{f \in \mathit{Arr} \mid (o, f) \in Y\}| < \infty \\
&\Rightarrow 0 < |\{f \in \mathit{Arr} \mid (o, f) \in Y\}| \wedge |\{f \in \mathit{Arr} \mid (o, f) \in X\}| < \infty \qquad (4.11)
\end{aligned}$$

With this we can then show that:

$$\begin{aligned}
&o \in ((\mathit{indexMods}(Y) \setminus \mathit{indexMods}(X)) \cup \mathit{finIndexMods}(X)) \\
\equiv{}& (0 < |\{f \in \mathit{Arr} \mid (o, f) \in Y\}| \wedge \neg(0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\}|)) \\
&\qquad \vee (0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\}| < \infty) \\
\equiv{}& (0 < |\{f \in \mathit{Arr} \mid (o, f) \in Y\}| \wedge |\{f \in \mathit{Arr} \mid (o, f) \in X\}| = 0) \vee (0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\}| < \infty) \\
\equiv{}& (\mathit{tt} \wedge |\{f \in \mathit{Arr} \mid (o, f) \in X\}| = 0) \vee (0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\}| < \infty) && \text{by (4.11)} \\
\equiv{}& |\{f \in \mathit{Arr} \mid (o, f) \in X\}| = 0 \vee (0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\}| < \infty) \\
\equiv{}& |\{f \in \mathit{Arr} \mid (o, f) \in X\}| < \infty \equiv \mathit{tt} && \text{by (4.11)}
\end{aligned}$$

□

Lemma 14. $|\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | \leq k$

Proof. By case distinction:

$\exists (o', f') \in X.\ f' \notin (\mathit{fs} \cup \mathit{Arr})$:

$$\begin{aligned}
|\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | &= |\ \mathit{finIndexMods}(\mathit{LS})\ | \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in \mathit{LS}\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid \mathit{tt}\} < \infty\}| \\
&= |\{o \mid 0 < |\mathit{Arr}| < \infty\}| \\
&= |\{o \mid 0 < \infty < \infty\}| = |\{o \mid \mathit{ff}\}| = |\emptyset| = 0 \leq k
\end{aligned}$$

$\forall (o', f') \in X.\ f' \in (\mathit{fs} \cup \mathit{Arr})$:

$$\begin{aligned}
&|\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | \\
={}& |\ \mathit{finIndexMods}(X \cup W^{N}_{\mathit{fs},n}(X) \cup W^{M}_{m}(X) \cup W^{K}_{k}(X))\ | \\
={}& |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in (X \cup W^{N}_{\mathit{fs},n}(X) \cup W^{M}_{m}(X) \cup W^{K}_{k}(X))\}| < \infty\}| \\
={}& |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^{N}_{\mathit{fs},n}(X) \\
&\qquad \vee (o, f) \in W^{M}_{m}(X) \vee (o, f) \in W^{K}_{k}(X)\}| < \infty\}|
\end{aligned}$$

We distinguish two cases:

$|\{o \mid \exists f \in \mathit{Arr}.\ (o, f) \in \mathit{ls}\}| > k$:

$$\begin{aligned}
&|\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \\
&\qquad\qquad \vee (o, f) \in W^M_m(X) \vee (o, f) \in W^K_k(X)\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \\
&\qquad\qquad \vee (o, f) \in W^M_m(X) \vee (o, f) \in (D^{\mathit{Object}} \times \mathit{Arr})\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \vee (o, f) \in W^M_m(X) \vee \mathit{tt}\}| < \infty\}| \\
&\qquad\qquad \text{as } f \in \mathit{Arr} \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid \mathit{tt}\}| < \infty\}| \\
&= |\{o \mid 0 < |\mathit{Arr}| < \infty\}| = |\{o \mid \mathit{ff}\}| = |\emptyset| = 0 \leq k
\end{aligned}$$

$|\{o \mid \exists f \in \mathit{Arr}.\ (o, f) \in \mathit{ls}\}| \leq k$:

$$\begin{aligned}
&|\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \\
&\qquad\qquad \vee (o, f) \in W^M_m(X) \vee (o, f) \in W^K_k(X)\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \\
&\qquad\qquad \vee (o, f) \in W^M_m(X) \vee (o, f) \in \emptyset\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \vee (o, f) \in W^M_m(X) \vee \mathit{ff}\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^N_{\mathit{fs},n}(X) \vee (o, f) \in W^M_m(X)\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X \vee (o, f) \in W^M_m(X)\}| < \infty\}| \qquad \text{by Lemma 10} \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}| \wedge \\
&\qquad |\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}| < \infty\}| \\
&= |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}|\} \cap \\
&\qquad \{o \mid\ |\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}| < \infty\}| \\
&\leq |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}|\}| \\
&= |\{o \mid (\{f \in \mathit{Arr} \mid (o, f) \in X\} \cup \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\}) \neq \emptyset\}| \\
&= |\{o \mid \{f \in \mathit{Arr} \mid (o, f) \in X\} \neq \emptyset \vee \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\} \neq \emptyset\}| \\
&= |\{o \mid \{f \in \mathit{Arr} \mid (o, f) \in X\} \neq \emptyset\} \cup \{o \mid \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\} \neq \emptyset\}\}|
\end{aligned}$$

Simplifying using Lemma 11 gives us

$$\begin{aligned}
&|\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | \\
&\leq |\{o \mid \{f \in \mathit{Arr} \mid (o, f) \in X\} \neq \emptyset\} \cup \{o \mid \{f \in \mathit{Arr} \mid (o, f) \in W^M_m(X)\} \neq \emptyset\}\}| \\
&= |\{o \mid \{f \in \mathit{Arr} \mid (o, f) \in X\} \neq \emptyset\}| \qquad \text{by Lemma 11} \\
&= |\{o \mid \exists f \in \mathit{Arr}.\ (o, f) \in X\}| \\
&\leq k \qquad \text{by case distinction assumption} \quad \square
\end{aligned}$$

Lemma 15. $\hat{f} \notin (\mathit{fs} \cup \mathit{Arr}) \wedge (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X) \rightarrow W_{\mathit{fs},n,m,k}(X) = \mathit{LS}$

Proof. By case distinction

$\exists (o', f') \in X.\ f' \notin (\mathit{fs} \cup \mathit{Arr})$: Then by definition we have our conclusion: $W_{\mathit{fs},n,m,k}(X) = \mathit{LS}$

$\forall (o', f') \in X.\ f' \in (\mathit{fs} \cup \mathit{Arr})$: This means in particular, that

$$(\hat{o}, \hat{f}) \notin X \tag{4.12}$$

Then we have a proof by contradiction, by assuming part of the premiss: $\hat{f} \notin (\mathit{fs} \cup \mathit{Arr})$

$$\begin{aligned}
(\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(X) &\equiv (\hat{o}, \hat{f}) \in (X \cup W^{N}_{\mathit{fs},n}(X) \cup W^{M}_{m}(X) \cup W^{K}_{k}(X)) \\
&\equiv (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^{N}_{\mathit{fs},n}(X) \vee (\hat{o}, \hat{f}) \in W^{M}_{m}(X) \vee (\hat{o}, \hat{f}) \in W^{K}_{k}(X) \\
&\equiv (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^{N}_{\mathit{fs},n}(X) \vee \mathit{ff} \vee \mathit{ff} && \text{by Lemma 3} \\
&\equiv (\hat{o}, \hat{f}) \in X \vee (\hat{o}, \hat{f}) \in W^{N}_{\mathit{fs},n}(X) \equiv (\hat{o}, \hat{f}) \in X \vee \mathit{ff} && \text{by Lemma 10} \\
&\equiv (\hat{o}, \hat{f}) \in X \equiv \mathit{ff} && \text{by (4.12)}
\end{aligned}$$

Which is a contradiction to the other premiss. □

Lemma 16. $0 \leq \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \leq (m+1)(k+1)$

Proof. By Lemma 14 we know that there is some $z \in \mathbb{N}$, such that

$$z = |\ \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))\ | \leq k \tag{4.13}$$

As $\mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X)) \subseteq \mathit{indexMods}(W_{\mathit{fs},n,m,k}(X))$ there is some $y \in (\mathbb{N} \cup \{\infty\})$ such that

$$y = |\ \mathit{indexMods}(W_{\mathit{fs},n,m,k}(X))\ | \geq z \tag{4.14}$$

$$\begin{aligned}
&\mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \\
&= (m+1)(k+1-\mathit{min}(k+1, |\ \mathit{indexMods}(W_{\mathit{fs},n,m,k}(X))\ |)) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o, f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&= (m+1)(k+1-\mathit{min}(k+1, y)) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o, f) \in W_{\mathit{fs},n,m,k}(X)\}| && \text{by (4.14)}
\end{aligned}$$

Simplifying inside the sum, we note that for any object o, in particular an $o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))$, by Lemma 12 either:

$\{f \in \mathit{Arr} \mid (o, f) \in W_{\mathit{fs},n,m,k}(X)\} = \mathit{Arr}$: This contradicts the guard $o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))$, as

$$\begin{aligned}
o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X)) &\equiv o \in \{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in W_{\mathit{fs},n,m,k}(X)\}| < \infty\} \\
&\equiv 0 < |\{f \in \mathit{Arr} \mid (o, f) \in W_{\mathit{fs},n,m,k}(X)\}| < \infty \\
&\equiv 0 < |\mathit{Arr}| < \infty \\
&\equiv 0 < \infty < \infty \\
&\equiv \mathit{ff}
\end{aligned}$$

$|\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(X)\}| \leq m$: We first show $0 \leq \mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X))$:

$$\begin{aligned}
&\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \\
&= (m+1)(k+1-\mathit{min}(k+1,y)) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&\geq (m+1)(k+1-(k+1)) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&= 0 + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&\geq 0 + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - m \\
&= 0 + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 \\
&\geq 0
\end{aligned}$$

Now we show $\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \leq (m+1)(k+1)$:

$$\begin{aligned}
&\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \\
&= (m+1)(k+1-\mathit{min}(k+1,y)) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&\leq (m+1)(k+1-\mathit{min}(k+1,y)) + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - 0 \\
&= (m+1)(k+1-\mathit{min}(k+1,y)) + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} m + 1 \\
&= (m+1)(k+1-\mathit{min}(k+1,y)) + z(m+1) && \text{by (4.13)} \\
&= (m+1)(k+1-\mathit{min}(k+1,y)+z)
\end{aligned}$$

By case distinction

$k+1 \leq y$:

$$\begin{aligned}
\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) &\leq (m+1)(k+1-\mathit{min}(k+1,y)+z) \\
&= (m+1)(k+1-(k+1)+z) \\
&= (m+1)z \\
&< (m+1)(k+1) && \text{by (4.13)}
\end{aligned}$$

$k+1 > y$:

$$\begin{aligned}
\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) &\leq (m+1)(k+1-\mathit{min}(k+1,y)+z) \\
&= (m+1)(k+1-y+z) \\
&= (m+1)(k+1+(z-y)) \\
&\leq (m+1)(k+1) && \text{by (4.14)} \quad \square
\end{aligned}$$

Lemma 17. $o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X)) \rightarrow 0 < 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \leq m+1$

Proof.

$$\begin{aligned} o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X)) &\equiv 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| < \infty \\ &\Rightarrow |\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| < \infty \\ &\Rightarrow \{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\} \neq \mathit{Arr} \\ &\Rightarrow |\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \leq m \qquad \text{by Lemma 12} \\ &\Rightarrow 0 < 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \leq m+1 \qquad \square \end{aligned}$$

Lemma 18. $|\ \mathit{indexMods}(W_{\mathit{fs},n,m,k}(Y))\ | > k \Rightarrow \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(Y)) = \emptyset$

Proof.

$$\begin{aligned} |\ \mathit{indexMods}(W_{\mathit{fs},n,m,k}(Y))\ | > k &\equiv |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}|\}| > k \\ &\equiv |\{o \mid \exists f \in \mathit{Arr}.\ (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}| > k \\ &\Rightarrow W_k^K(Y) = (D^{\mathit{Object}} \times \mathit{Arr}) \\ &\Rightarrow W_{\mathit{fs},n,m,k}(Y) \supseteq (D^{\mathit{Object}} \times \mathit{Arr}) \\ &\Rightarrow \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(Y)) = \emptyset \qquad \square \end{aligned}$$

Lemma 19.

$$W_{\mathit{fs},n,m,k}(X) \subseteq W_{\mathit{fs},n,m,k}(Y) \rightarrow \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \geq \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(Y))$$

Proof. As premiss we have

$$W_{\mathit{fs},n,m,k}(X) \subseteq W_{\mathit{fs},n,m,k}(Y) \tag{4.15}$$

Let $x_{all}, y_{all} \in (\mathbb{N} \cup \{\infty\})$ be defined as

$$\begin{aligned} x_{all} &:= |\ \mathit{indexMods}(W_{\mathit{fs},n,m,k}(X))\ | = |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}|\}| \\ y_{all} &:= |\ \mathit{indexMods}(W_{\mathit{fs},n,m,k}(Y))\ | = |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}|\}| \end{aligned}$$

By subset relation we know that

$$x_{all} \leq y_{all} \tag{4.16}$$

By case distinction

$y_{all} > k$:

$$\begin{aligned} &\mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \\ &= (m+1)(k+1-\mathit{min}(k+1,x_{all})) \\ &\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\ &\geq (m+1)(k+1-\mathit{min}(k+1,x_{all})) + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-m \qquad \text{by Lemma 12} \\ &\geq (m+1)(k+1-\mathit{min}(k+1,x_{all})) + 0 \\ &\geq (m+1)(k+1-\mathit{min}(k+1,y_{all})) + 0 \qquad \text{by (4.16)} \\ &= (m+1)(k+1-\mathit{min}(k+1,y_{all})) + \sum_{o \in \emptyset} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}| \\ &= (m+1)(k+1-\mathit{min}(k+1,y_{all})) \\ &\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(Y))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}| \qquad \text{by Lemma 18} \\ &= \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(Y)) \end{aligned}$$

$y_{\mathit{all}} \leq k$:

$$
\begin{aligned}
&\mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \\
&= (m+1)(k+1-\mathit{min}(k+1, x_{\mathit{all}})) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&= (m+1)(k+1-x_{\mathit{all}}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| && \text{by (4.16)} \\
&= (m+1)(k+1+0-x_{\mathit{all}}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&= (m+1)(k+1+(y_{\mathit{all}}-y_{\mathit{all}})-x_{\mathit{all}}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&= (m+1)(k+1-y_{\mathit{all}}+(y_{\mathit{all}}-x_{\mathit{all}})) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&= (m+1)(k+1-y_{\mathit{all}})+(m+1)(y_{\mathit{all}}-x_{\mathit{all}}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&= (m+1)(k+1-y_{\mathit{all}}) + \sum_{o \in (\mathit{indexMods}(W_{\mathit{fs},n,m,k}(Y)) \setminus \mathit{indexMods}(W_{\mathit{fs},n,m,k}(X)))} m+1 \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| && \text{by (4.15)} \\
&\geq (m+1)(k+1-y_{\mathit{all}}) \\
&\quad + \sum_{o \in (\mathit{indexMods}(W_{\mathit{fs},n,m,k}(Y)) \setminus \mathit{indexMods}(W_{\mathit{fs},n,m,k}(X)))} m+1-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(X))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
&\geq (m+1)(k+1-y_{\mathit{all}}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(Y))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(X)\}| \\
& && \text{by Lemmas 13 and 17} \\
&\geq (m+1)(k+1-y_{\mathit{all}}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(Y))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}| && \text{by (4.15)} \\
&= (m+1)(k+1-\mathit{min}(k+1, y_{\mathit{all}})) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{\mathit{fs},n,m,k}(Y))} 1+m-|\{f \in \mathit{Arr} \mid (o,f) \in W_{\mathit{fs},n,m,k}(Y)\}| \\
&= \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(Y)) && \square
\end{aligned}
$$

Lemma 20. $\hat{f} \in \mathit{Arr} \wedge (\hat{o}, \hat{f}) \notin W_{fs,n,m,k}(X) \wedge W_{fs,n,m,k}(X) \subset W_{fs,n,m,k}(Y) \wedge (\hat{o}, \hat{f}) \in W_{fs,n,m,k}(Y) \rightarrow \mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) > \mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(Y))$

Proof. As premisses we have

$$\hat{f} \in \mathit{Arr} \tag{4.17}$$

$$(\hat{o}, \hat{f}) \notin W_{fs,n,m,k}(X) \tag{4.18}$$

$$W_{fs,n,m,k}(X) \subset W_{fs,n,m,k}(Y) \tag{4.19}$$

$$(\hat{o}, \hat{f}) \in W_{fs,n,m,k}(Y) \tag{4.20}$$

As in Lemma 19, let $x_{all}, y_{all} \in (\mathbb{N} \cup \{\infty\})$ be defined as

$$\begin{aligned} x_{all} &:= |\ \mathit{indexMods}(W_{fs,n,m,k}(X))\ | = |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\}| \\ y_{all} &:= |\ \mathit{indexMods}(W_{fs,n,m,k}(Y))\ | = |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}|\}| \end{aligned}$$

By subset relation we know that

$$x_{all} \leq y_{all} \tag{4.21}$$

We know that

$$\begin{aligned} (\hat{o}, \hat{f}) \notin W_{fs,n,m,k}(X) &\Rightarrow (D^{Object} \times \mathit{Arr}) \nsubseteq W_{fs,n,m,k}(X) && \text{by (4.17)} \\ &\Rightarrow W_k^K(X) \neq (D^{Object} \times \mathit{Arr}) \\ &\equiv |\{o \mid \exists f \in \mathit{Arr}.\ (o,f) \in W_{fs,n,m,k}(X)\}| \leq k \\ &\equiv |\{o \mid 0 < |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\}| \leq k \\ &\equiv |\ \mathit{indexMods}(W_{fs,n,m,k}(X))\ | \leq k \\ &\equiv x_{all} \leq k && (4.22) \end{aligned}$$

Therefore

$$\begin{aligned} &\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \\ &= (m+1)(k+1-\mathit{min}(k+1, x_{all})) \\ &\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\ &= (m+1)(k+1-x_{all}) \\ &\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.22)} \end{aligned}$$

By case distinction either $y_{all} > k$ or $y_{all} \leq k$:

$y_{all} > k$:

$$\begin{aligned} &\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \\ &= (m+1)(k+1-x_{all}) + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\ &\geq (m+1)(k+1-x_{all}) + 0 && \text{by Lemma 12} \\ &> (m+1)(k+1-(k+1)) + 0 && \text{by (4.22)} \\ &= (m+1)(k+1-\mathit{min}(k+1, y_{all})) + 0 && \text{by case distinction assumption} \\ &= (m+1)(k+1-\mathit{min}(k+1, y_{all})) + \sum_{o \in \emptyset} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}| \\ &= (m+1)(k+1-\mathit{min}(k+1, y_{all})) \\ &\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}| && \text{by Lemma 18} \\ &= \mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(Y)) \end{aligned}$$

$y_{all} \leq k$: We know that

$$\begin{aligned}(\hat{o}, \hat{f}) \in W_{fs,n,m,k}(Y) \Rightarrow \hat{o} &\in \{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(Y)\}|\} \\ &\equiv \hat{o} \in \mathit{indexMods}(W_{fs,n,m,k}(Y)) \end{aligned} \tag{4.23}$$

Reformulating $\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X))$ gives us

$$\begin{aligned}
&\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \\
&= (m+1)(k+1-x_{all}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&= (m+1)(k+1+0-x_{all}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&= (m+1)(k+1+(y_{all}-y_{all})-x_{all}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&= (m+1)(k+1-y_{all}+(y_{all}-x_{all})) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&= (m+1)(k+1-y_{all}) + (m+1)(y_{all}-x_{all}) \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&= (m+1)(k+1-y_{all}) + \sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1 \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.19)}
\end{aligned}$$

It remains to show that the sum of the two sums is greater than

$$\sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}| \tag{4.24}$$

With this we can then finish the proof:

$$\begin{aligned}
&\mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(X)) \\
&= (m+1)(k+1-y_{all}) + \sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1 \\
&\quad + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| \\
&> (m+1)(k+1-y_{all}) + \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}| \\
&= \mathit{indexModsLeft}_{m,k}(W_{fs,n,m,k}(Y))
\end{aligned}$$

We consider the following three cases and show in each that the sum of the two sums is greater than (4.24):

Case 1: $\hat{o} \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))$ and $\hat{o} \notin \mathit{finIndexMods}(W_{fs,n,m,k}(Y))$. We know that

$$\begin{aligned}
&\mathit{finIndexMods}(W_{fs,n,m,k}(Y))\\
\subset\ &\mathit{finIndexMods}(W_{fs,n,m,k}(Y)) \cup \{\hat{o}\} && \text{by case distinction assumption}\\
\subseteq\ &(\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X))) \cup \mathit{finIndexMods}(W_{fs,n,m,k}(X)) \cup \{\hat{o}\}\\
& && \text{by (4.19) and Lemma 13}\\
=\ &(\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X))) \cup \mathit{finIndexMods}(W_{fs,n,m,k}(X))
\end{aligned}$$

It therefore holds that

$$\begin{aligned}
&\mathit{finIndexMods}(W_{fs,n,m,k}(Y)) \qquad (4.25)\\
\subset\ &((\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X))) \cup \mathit{finIndexMods}(W_{fs,n,m,k}(X)))
\end{aligned}$$

With this we can now show that

$$\begin{aligned}
&\sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\\
\geq\ &\sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1 - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\\
>\ &\sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| && \text{by (4.25) and L. 17}\\
\geq\ &\sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}| && \text{by (4.19)}
\end{aligned}$$

Case 2: $\hat{o} \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))$.

$$\begin{aligned}
&|\{f \in \mathit{Arr} \mid (\hat{o},f) \in W_{fs,n,m,k}(X)\}|\\
<\ &|\{f \in \mathit{Arr} \mid (\hat{o},f) \in W_{fs,n,m,k}(X)\} \cup \{\hat{f}\}| && \text{by (4.18)}\\
\leq\ &|\{f \in \mathit{Arr} \mid (\hat{o},f) \in W_{fs,n,m,k}(Y)\}| && \text{by (4.19) and (4.20)}\\
\leq\ &m && \text{by Lemma 17}
\end{aligned}$$

Therefore we have the inequalities

$$|\{f \in \mathit{Arr} \mid (\hat{o},f) \in W_{fs,n,m,k}(X)\}| < |\{f \in \mathit{Arr} \mid (\hat{o},f) \in W_{fs,n,m,k}(Y)\}| \leq m \qquad (4.26)$$

$$\begin{aligned}
&\sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\\
\geq\ &\sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1 - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}|\\
\geq\ &\sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(X)\}| && \text{by L. 13 and L. 17}\\
>\ &\sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o,f) \in W_{fs,n,m,k}(Y)\}|\\
& && \text{by (4.19) and (4.26)}
\end{aligned}$$

Case 3: $\hat{o} \notin \mathit{finIndexMods}(W_{fs,n,m,k}(X))$. Reformulating this yields:

$$\begin{aligned}
&\hat{o} \notin \mathit{finIndexMods}(W_{fs,n,m,k}(X))\\
&\equiv \hat{o} \notin \{o \mid 0 < |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(X)\}| < \infty\}\\
&\equiv |\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| = 0 \vee |\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| = \infty
\end{aligned}$$

Which we can express as the implication:

$$|\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| < \infty \to |\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| = 0 \quad (4.27)$$

Reformulating the premiss (4.18) gives us:

$$\begin{aligned}
(\hat{o}, \hat{f}) \notin W_{fs,n,m,k}(X) &\Rightarrow \{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\} \neq \mathit{Arr}\\
&\equiv |\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| \leq m && \text{by Lemma 12}\\
&\Rightarrow |\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| < \infty\\
&\equiv |\{f \in \mathit{Arr} \mid (\hat{o}, f) \in W_{fs,n,m,k}(X)\}| = 0 && \text{by (4.27)}\\
&\equiv \hat{o} \notin \mathit{indexMods}(W_{fs,n,m,k}(X)) && (4.28)
\end{aligned}$$

By (4.23) and (4.28) we know that

$$\hat{o} \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X))) \quad (4.29)$$

We can therefore show that

$$\begin{aligned}
&\sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(X)\}|\\
&> \sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1 - |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(Y)\}|\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(X)\}|\\
&&\text{by (4.29) and (4.23)}\\
&\geq \sum_{o \in (\mathit{indexMods}(W_{fs,n,m,k}(Y)) \setminus \mathit{indexMods}(W_{fs,n,m,k}(X)))} m+1 - |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(Y)\}|\\
&+ \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(X))} 1 + m - |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(Y)\}| &&\text{by (4.19)}\\
&\geq \sum_{o \in \mathit{finIndexMods}(W_{fs,n,m,k}(Y))} 1 + m - |\{f \in \mathit{Arr} \mid (o, f) \in W_{fs,n,m,k}(Y)\}|\\
&&\text{by Lemmas 13 and 17} \quad \square
\end{aligned}$$

Similar to Lemma 9, this (with Lemma 19) is the main lemma of this subsection, showing that as more array index field modifications exist after a widening, the number of array index modifications available decreases, i.e. with Lemma 16 that the number of such steps is bound.

We introduce a final helper function:

$$\begin{aligned} \mathit{modsLeft}_{\mathit{fs},n,m,k} : A^{LS} &\to \mathbb{N} \\ \bot &\mapsto 2 + |\mathit{fs}|(n+1) + (m+1)(k+1) \\ \mathit{LS} &\mapsto 0 \\ \mathit{ls} &\mapsto 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(\mathit{ls}) + \mathit{indexModsLeft}_{m,k}(\mathit{ls}) \end{aligned}$$

Intuitively, $\mathit{modsLeft}_{\mathit{fs},n,m,k}$ expresses the total number of widening steps left for any new modification of the location set. The result of $\mathit{modsLeft}_{\mathit{fs},n,m,k}(X)$ is an upper bound on the maximal number of iterations the following algorithm can have:

1. Set X to $W_{\mathit{fs},n,m,k}(X \cup (o,f))$, where $(o,f) \notin X$.
2. If $X \neq \mathit{LS}$, goto 1.

Lemma 21. $0 \leq \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) \leq 1 + |\mathit{fs}|(n+1) + (m+1)(k+1)$

Proof. If $W_{\mathit{fs},n,m,k}(X) = \mathit{LS}$, we have

$$\mathit{modsLeft}_{\mathit{fs},n,m,k}(\mathit{LS}) = 0 \leq 1 + |\mathit{fs}|(n+1) + (m+1)(k+1)$$

Otherwise, we show first that $0 \leq \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X))$:

$$\begin{aligned} \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) &= 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \\ &\geq 1 + 0 + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by Lemma 7} \\ &\geq 1 + 0 + 0 && \text{by Lemma 16} \\ &> 0 \end{aligned}$$

Next we show $\mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) \leq 1 + |\mathit{fs}|(n+1) + (m+1)(k+1)$:

$$\begin{aligned} \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) &= 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \\ &\leq 1 + |\mathit{fs}|(n+1) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by Lemma 7} \\ &\leq 1 + |\mathit{fs}|(n+1) + (m+1)(k+1) && \text{by Lemma 16} \quad \square \end{aligned}$$

Lemma 22. $\nabla_{\mathit{fs},n,m,k}$ *is commutative*

Proof.

$$\begin{aligned} x \nabla_{\mathit{fs},n,m,k}\, y &= \begin{cases} \bot & \text{, if } x \sqcup^{LS} y = \bot \\ W_{\mathit{fs},n,m,k}(x \sqcup^{LS} y) & \text{, otherwise} \end{cases} \\ &= \begin{cases} \bot & \text{, if } y \sqcup^{LS} x = \bot \\ W_{\mathit{fs},n,m,k}(y \sqcup^{LS} x) & \text{, otherwise} \end{cases} && \text{by Lemma 1} \\ &= y \nabla_{\mathit{fs},n,m,k}\, x && \square \end{aligned}$$

Lemma 23.

$$W_{\mathit{fs},n,m,k}(X) \sqsubset^{LS} W_{\mathit{fs},n,m,k}(Y) \rightarrow \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) > \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(Y))$$

Proof. Rephrasing the premiss gives us

$$\begin{aligned}
& W_{\mathit{fs},n,m,k}(X) \sqsubset^{LS} W_{\mathit{fs},n,m,k}(Y) \\
\equiv\ & W_{\mathit{fs},n,m,k}(X) \neq W_{\mathit{fs},n,m,k}(Y) \wedge W_{\mathit{fs},n,m,k}(X) \subseteq W_{\mathit{fs},n,m,k}(Y) \\
\equiv\ & W_{\mathit{fs},n,m,k}(X) \subset W_{\mathit{fs},n,m,k}(Y)
\end{aligned} \tag{4.30}$$

It follows that

$$\forall \mathit{loc} \in LS.\ \mathit{loc} \in W_{\mathit{fs},n,m,k}(X) \rightarrow \mathit{loc} \in W_{\mathit{fs},n,m,k}(Y) \tag{4.31}$$

$$\exists \mathit{loc} \in LS.\ \mathit{loc} \notin W_{\mathit{fs},n,m,k}(X) \wedge \mathit{loc} \in W_{\mathit{fs},n,m,k}(Y) \tag{4.32}$$

We therefore know that $W_{\mathit{fs},n,m,k}(X) \neq LS$ and therefore

$$\mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) = 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) \tag{4.33}$$

By case analysis

$W_{\mathit{fs},n,m,k}(Y) = LS$:

$$\begin{aligned}
& \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) && \\
=\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by (4.33)} \\
\geq\ & 1 + 0 + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by Lemma 7} \\
\geq\ & 1 + 0 + 0 && \text{by Lemma 16} \\
>\ & 0 && \\
=\ & \mathit{modsLeft}_{\mathit{fs},n,m,k}(LS) && \\
=\ & \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(Y)) &&
\end{aligned}$$

$W_{\mathit{fs},n,m,k}(Y) \neq LS$: By (4.32) there is some $(\hat{o}, \hat{f}) \in LS$, such that

$$(\hat{o}, \hat{f}) \notin W_{\mathit{fs},n,m,k}(X) \wedge (\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(Y) \tag{4.34}$$

We further differentiate between 3 cases:

$\hat{f} \notin (\mathit{fs} \cup \mathit{Arr})$: This leads to the contradiction

$$\begin{aligned}
(\hat{o}, \hat{f}) \in W_{\mathit{fs},n,m,k}(Y) &\equiv W_{\mathit{fs},n,m,k}(Y) = LS && \text{by Lemma 15} \\
&\equiv \mathit{ff} && \text{by case analysis assumption}
\end{aligned}$$

$\hat{f} \in \mathit{fs}$:

$$\begin{aligned}
& \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) && \\
=\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by (4.33)} \\
\geq\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(Y)) && \text{by Lemma 19} \\
>\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(Y)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(Y)) && \text{by Lemma 9} \\
=\ & \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(Y)) &&
\end{aligned}$$

$\hat{f} \in \mathit{Arr}$:

$$\begin{aligned}
& \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(X)) && \\
=\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(X)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by (4.33)} \\
\geq\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(Y)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(X)) && \text{by Lemma 8} \\
>\ & 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(W_{\mathit{fs},n,m,k}(Y)) + \mathit{indexModsLeft}_{m,k}(W_{\mathit{fs},n,m,k}(Y)) && \text{by Lemma 20} \\
=\ & \mathit{modsLeft}_{\mathit{fs},n,m,k}(W_{\mathit{fs},n,m,k}(Y)) &&
\end{aligned}$$

□

Lemma 24. $X \subseteq Y \rightarrow X \subseteq W_{\mathit{fs},n,m,k}(Y)$

Proof. This is trivially the case, as by definition either:

1. $W_{\mathit{fs},n,m,k}(Y) = \mathit{LS}$ and obviously $X \subseteq \mathit{LS}$, or
2. $W_{\mathit{fs},n,m,k}(Y) = Y \cup W^{N}_{\mathit{fs},n}(Y) \cup W^{M}_{m}(Y) \cup W^{K}_{k}(Y)$, which contains all elements of Y and therefore also all elements of X. □

With these lemmas and helper functions we can now finally prove that $\nabla_{\mathit{fs},n,m,k}$ is a widening operator:

Theorem 1. *$\nabla_{\mathit{fs},n,m,k}$ is a widening operator*

Proof. We must show the following:

$x \sqsubseteq^{LS} (x \nabla_{\mathit{fs},n,m,k}\ y)$ and $y \sqsubseteq^{LS} (x \nabla_{\mathit{fs},n,m,k}\ y)$: By Lemma 22, $\nabla_{\mathit{fs},n,m,k}$ is commutative, so it suffices to only show that $x \sqsubseteq^{LS} (x \nabla_{\mathit{fs},n,m,k}\ y)$. By case distinction we show this for the following three cases:

$x = \bot$:

$$x \sqsubseteq^{LS} (x \nabla_{\mathit{fs},n,m,k}\ y) = \mathit{tt}$$

$x \neq \bot$ and $y = \bot$:

$$\begin{aligned} x \sqsubseteq^{LS} (x \nabla_{\mathit{fs},n,m,k}\ y) &= x \sqsubseteq^{LS} W_{\mathit{fs},n,m,k}(x \sqcup^{LS} y) \\ &= x \sqsubseteq^{LS} W_{\mathit{fs},n,m,k}(x) \\ &= x \subseteq W_{\mathit{fs},n,m,k}(x) \\ &= \mathit{tt} && \text{by Lemma 24} \end{aligned}$$

$x \neq \bot$ and $y \neq \bot$:

$$\begin{aligned} x \sqsubseteq^{LS} (x \nabla_{\mathit{fs},n,m,k}\ y) &= x \sqsubseteq^{LS} W_{\mathit{fs},n,m,k}(x \sqcup^{LS} y) \\ &= x \sqsubseteq^{LS} W_{\mathit{fs},n,m,k}(x \cup y) \\ &= x \subseteq W_{\mathit{fs},n,m,k}(x \cup y) \\ &= \mathit{tt} && \text{by Lemma 24} \end{aligned}$$

A sequence produced by $\nabla_{\mathit{fs},n,m,k}$ is ultimately stationary: Given any sequence $\langle y'_z \rangle$, we must prove that the sequence $\langle x'_z \rangle$ defined by

$$\begin{aligned} x'_0 &= \bot \\ x'_{z+1} &= (x'_z \quad \nabla_{\mathit{fs},n,m,k} \quad y'_z) \end{aligned}$$

is ultimately stationary. Proof by contradiction:

We know that $x'_i \sqsubseteq x'_{i+1}$. Further, if $\langle x'_z \rangle$ were not ultimately stationary, there must exist an infinite sequence $\langle i'_z \rangle$, such that $i_0 = 0$, $i_j < i_{j+1}$ and $x'_{i_j} = x'_{i_{j+1}-1} \sqsubset^{LS} x'_{i_{j+1}}$. We show that this sequence cannot exist by limiting the number of elements in any such sequence to a maximum of $3 + |\mathit{fs}|(n+1) + (m+1)(k+1)$. This is the case because

1. $\mathit{modsLeft}_{\mathit{fs},n,m,k}(x'_{i_0}) = 2 + |\mathit{fs}|(n+1) + (m+1)(k+1)$
   By definition

$$\begin{aligned} \mathit{modsLeft}_{\mathit{fs},n,m,k}(x'_{i_0}) &= \begin{cases} 2 + |\mathit{fs}|(n+1) + (m+1)(k+1) & \text{, if } x'_{i_0} = \bot \\ 0 & \text{, if } x'_{i_0} = \mathit{LS} \\ 1 + \mathit{fieldModsLeft}_{\mathit{fs},n}(x'_{i_0}) + \mathit{indexModsLeft}_{m,k}(x'_{i_0}) & \text{, otherwise} \end{cases} \\ &= 2 + |\mathit{fs}|(n+1) + (m+1)(k+1) && \text{as } x'_{i_0} = x'_0 = \bot \end{aligned}$$

2. $\forall j \in \mathbb{N}.\ \mathit{modsLeft}_{fs,n,m,k}(x'_{i_j}) > \mathit{modsLeft}_{fs,n,m,k}(x'_{i_{j+1}})$
   By case distinction either

   $j = 0$: We know

$$x'_{i_j} = \bot \tag{4.35}$$

As $x'_{i_j} = \bot \sqsubset^{LS} x'_{i_{j+1}}$, we also know

$$x'_{i_{j+1}} \neq \bot \tag{4.36}$$

We use this to show

$$\begin{aligned}
x'_{i_{j+1}} \neq \bot &\equiv (x'_{i_{j+1}-1} \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \neq \bot \\
&\equiv (x'_{i_j} \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \neq \bot \\
&\equiv (\bot \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \neq \bot \\
&\equiv (\bot \sqcup^{LS} y'_{i_{j+1}-1}) \neq \bot \\
&\equiv y'_{i_{j+1}-1} \neq \bot
\end{aligned} \tag{4.37}$$

Therefore

$$\begin{aligned}
\mathit{modsLeft}_{fs,n,m,k}(x'_{i_j}) &= \mathit{modsLeft}_{fs,n,m,k}(\bot) \\
&= 2 + |fs|(n+1) + (m+1)(k+1) \\
&> 1 + |fs|(n+1) + (m+1)(k+1) \\
&\geq \mathit{modsLeft}_{fs,n,m,k}(W_{fs,n,m,k}(y'_{i_{j+1}-1})) && \text{by Lemma 21 and (4.37)} \\
&= \mathit{modsLeft}_{fs,n,m,k}(W_{fs,n,m,k}(\bot \sqcup^{LS} y'_{i_{j+1}-1})) \\
&= \mathit{modsLeft}_{fs,n,m,k}(\bot \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \\
&= \mathit{modsLeft}_{fs,n,m,k}(x'_{i_j} \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \\
&= \mathit{modsLeft}_{fs,n,m,k}(x'_{i_{j+1}-1} \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \\
&= \mathit{modsLeft}'_{fs,n,m,k}(x'_{i_{j+1}})
\end{aligned}$$

$j > 0$: As $\bot \sqsubset^{LS} \ldots \sqsubset^{LS} x'_{i_j} \sqsubset^{LS} x'_{i_{j+1}}$ we know

$$x'_{i_j} \neq \bot \tag{4.38}$$

$$x'_{i_{j+1}} \neq \bot \tag{4.39}$$

Therefore

$$\begin{aligned}
& x'_{i_j} \sqsubset^{LS} x'_{i_{j+1}} \\
\equiv\ & (x_{i_j-1} \ \nabla_{fs,n,m,k}\ y_{i_j-1}) \sqsubset^{LS} (x'_{i_{j+1}-1} \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) \\
\equiv\ & W_{fs,n,m,k}(x_{i_j-1} \sqcup^{LS} y_{i_j-1}) \sqsubset^{LS} (x'_{i_{j+1}-1} \ \nabla_{fs,n,m,k}\ y'_{i_{j+1}-1}) && \text{by (4.38)} \\
\equiv\ & W_{fs,n,m,k}(x_{i_j-1} \sqcup^{LS} y_{i_j-1}) \sqsubset^{LS} W_{fs,n,m,k}(x'_{i_{j+1}-1} \sqcup^{LS} y'_{i_{j+1}-1}) && \text{by (4.39)} \\
\Rightarrow\ & \mathit{modsLeft}_{fs,n,m,k}(W_{fs,n,m,k}(x_{i_j-1} \sqcup^{LS} y_{i_j-1})) \ > \\
& \mathit{modsLeft}_{fs,n,m,k}(W_{fs,n,m,k}(x'_{i_{j+1}-1} \sqcup^{LS} y'_{i_{j+1}-1})) && \text{by Lemma 23} \\
\equiv\ & \mathit{modsLeft}_{fs,n,m,k}(x'_{i_j}) > \mathit{modsLeft}_{fs,n,m,k}(x'_{i_{j+1}})
\end{aligned}$$

3. $\forall j \in \mathbb{N}.\ \mathit{modsLeft}_{fs,n,m,k}(x'_{i_j}) \geq 0$
   By case distinction
   either $x'_{i_j} = \bot$:

$$\begin{aligned} \mathit{modsLeft}_{fs,n,m,k}(x'_{i_j}) &= \mathit{modsLeft}_{fs,n,m,k}(\bot) \\ &= 2 + |fs|(n+1) + (m+1)(k+1) \\ &\geq 0 \end{aligned}$$

   or $x'_{i_j} \neq \bot$:

$$\begin{aligned} \mathit{modsLeft}_{fs,n,m,k}(x'_{i_j}) &= \mathit{modsLeft}_{fs,n,m,k}(x_{i_j - 1} \nabla_{fs,n,m,k} y_{i_j - 1}) \\ &= \mathit{modsLeft}_{fs,n,m,k}(W_{fs,n,m,k}(x_{i_j - 1} \sqcup^{LS} y_{i_j - 1})) \\ &\geq 0 && \text{by Lemma 21} \quad \square \end{aligned}$$

4.3 Abstraction For All Heaps

Although normal form heaps built on an initial heap *old* are the only ones of particular interest to us, we can extend the heap abstraction to cover all heaps by introducing a top element $\top$ to hold all heaps which are not normal form heaps built on *old*:

$$\begin{aligned} \mathscr{A}^{\mathit{Heap}} &= (A^{\mathit{Heap}}, \sqcup^{\mathit{Heap}}, \sqsubseteq^{\mathit{Heap}}) \\ A^{\mathit{Heap}} &= \{\top\} \cup A^{LS} \\ x \sqcup^{\mathit{Heap}} y &= \begin{cases} \top & \text{, if } x = \top \text{ or } y = \top \\ x \sqcup^{LS} y & \text{, otherwise} \end{cases} \\ x \sqsubseteq^{\mathit{Heap}} y &= \begin{cases} \mathit{tt} & \text{, if } y = \top \\ \mathit{ff} & \text{, if } x = \top \text{ and } y \neq \top \\ x \sqsubseteq^{LS} y & \text{, otherwise} \end{cases} \end{aligned}$$

With this abstract domain, we can now define families of abstraction and concretization functions between $2^{D^{\mathit{Heap}}}$ and A^{LS}. For all $\mathit{old} \in \mathsf{Trm}_{\mathit{Heap}}$ and $h \in D^{\mathit{Heap}}$, in particular for h the value of *old* in a valid model, we define the families of abstraction and concretization functions as:

$$\begin{aligned} \alpha_h^{\mathit{old}} : 2^{D^{\mathit{Heap}}} &\to A^{LS} \\ \emptyset &\mapsto \bot \\ \mathit{heaps} &\mapsto \begin{cases} \top & \text{, if } \mathit{heaps} \not\subseteq D_h^{\mathit{Heap}_{\mathit{old}}^{NF}} \\ \{(o,f) \mid f \neq \mathit{created}^{\mathscr{M}} \wedge \exists h' \in \mathit{heaps}.\ h'(o,f) \neq h(o,f)\} & \text{, otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \gamma_h^{\mathit{old}} : A^{LS} &\to 2^{D^{\mathit{Heap}}} \\ \bot &\mapsto \emptyset \\ \top &\mapsto D^{\mathit{Heap}} \\ \mathit{ls} &\mapsto \{h' \mid \forall o \in D^{\mathit{Object}}.\ (h(o, \mathit{created}^{\mathscr{M}}) = \mathit{tt} \to h'(o, \mathit{created}^{\mathscr{M}}) = \mathit{tt}) \\ &\qquad \wedge\ \forall f \in (D^{\mathit{Field}} \setminus \{\mathit{created}^{\mathscr{M}}\}).\ (o,f) \notin \mathit{ls} \to h'(o,f) = h(o,f)\} \end{aligned}$$

4.3.1 Syntactic Representation of Abstract Heaps

While generating specifications, we encounter modified heap terms as updates to the program variable `heap`. If we wish to join two updates where the value for `heap` is not identical, we need to abstract these heap terms (and thus all possible semantic heaps they can express) and then join the resulting abstractions, possibly widening the result thereof. We can accomplish an overapproximation of this purely on the syntactic heap terms, by performing the following steps:

1. Find the term $\mathit{old} \in \mathsf{Trm}_{\mathit{Heap}}$ which represents the state of the heap before the loop or method call.
2. Abstract the heap terms we wish to join, by determining a location set term for each heap term, overapproximating all locations which may have been changed in regards to old.
3. Perform a syntactic union operation on the resulting location set terms.
4. Perform syntactic widening along the lines of the widening operator $\nabla_{fs,n,m,k}$.

Example 4. *We consider the following initial sequent, containing a program traversing a list (of* `final` *class* `List`*) and replacing all values with* 0*:*

$$\Longrightarrow \{\mathtt{l} := l \parallel \mathtt{heap} := h_1\}[\mathtt{while\ (l\ !=\ null)\ \{\ l.value\ =\ 0;\ l\ =\ l.next;\ \}}]\phi$$

Symbolic execution of a single loop iteration leads to the sequent:

$$l \neq \mathit{null} \Longrightarrow \{\mathtt{l} := \mathit{select}_{\mathit{List}}(h_1, l, \mathtt{next}) \parallel \mathtt{heap} := \mathit{store}(h_1, l, \mathtt{value}, 0)\} \\ [\mathtt{while\ (l\ !=\ null)\ \{\ l.value\ =\ 0;\ l\ =\ l.next;\ \}}]\phi$$

As the two heap terms h_1 and $\mathit{store}(h_1, l, \mathtt{value}, 0)$ are not identical, we wish to abstract and join them before evaluating another loop iteration. The first step is therefore to determine the heap term old, representing the state of the heap before the loop. For this example, that means:

$$\mathit{old} = h_1 \tag{4.40}$$

In order to determine if a term $h \in \mathsf{Trm}_{\mathit{Heap}}$ was built on old and if so to construct a location set term containing all locations that may change, we define the following family of functions:

$$\mathit{locs}_{\mathit{old}} : (\mathsf{Trm}_{\mathit{Heap}} \times \mathsf{Trm}_{\mathit{LocSet}}) \to (\mathsf{Trm}_{\mathit{LocSet}} \cup \{\top\})$$

$$(h, \mathit{ls}) \mapsto \begin{cases} \mathit{ls} & \text{, if } h = \mathit{old} \\ \mathit{locs}_{\mathit{old}}(h', \mathit{singleton}(o, f) \mathbin{\dot{\cup}} \mathit{ls}) & \text{, if } h = \mathit{store}(h', o, f, v) \neq \mathit{old} \\ \mathit{locs}_{\mathit{old}}(h', \mathit{ls}' \mathbin{\dot{\cup}} \mathit{ls}) & \text{, if } h = \mathit{anon}(h', \mathit{ls}', h'') \neq \mathit{old} \\ \mathit{locs}_{\mathit{old}}(h', \mathit{ls}) & \text{, if } h = \mathit{create}(h', o) \neq \mathit{old} \\ \top & \text{, otherwise} \end{cases}$$

Intuitively, the result of $\mathit{locs}_{\mathit{old}}$ (called with second parameter $\dot{\emptyset}$) represents an abstract element in A^{Heap} which overapproximates the heap, where a result of $\top$ represents $\top \in A^{\mathit{Heap}}$ and a location set term ls represents the abstract element for the value of that location set. As can be seen, *create* does not add to the location set term, as $\mathit{created}^{\mathscr{M}}$ is not a valid field in LS, while *store* and *anon* add terms reflecting the location sets they may have modified.

Example 5. *Continuing Example 4, we now wish to abstract and syntactically join the heap terms h_1 and $store(h_1, l, \texttt{value}, 0)$. From (4.40), abstracting means calculating:*

$$locs_{h_1}(h_1, \dot{\emptyset}) = \dot{\emptyset}$$
$$locs_{h_1}(store(h_1, l, \texttt{value}, 0), \dot{\emptyset}) = locs_{h_1}(h_1, singleton(l, \texttt{value}) \dot{\cup} \dot{\emptyset}) = singleton(l, \texttt{value}) \dot{\cup} \dot{\emptyset}$$

Syntactic joining results in:

$$locs_{h_1}(h_1, \dot{\emptyset}) \dot{\cup} locs_{h_1}(store(h_1, l, \texttt{value}, 0), \dot{\emptyset}) = \dot{\emptyset} \dot{\cup} singleton(l, \texttt{value}) \dot{\cup} \dot{\emptyset}$$

Which can be simplified to:

$$singleton(l, \texttt{value})$$

An overapproximative widening can be performed syntactically on the resulting location set term ls, by using $allLocs$, $allFields$ and $allObjects$ to widen locations if it cannot be proven that the semantic location sets represented by the terms would not be widened by $\nabla_{fs,n,m,k}$. This can be accomplished by starting side proofs, leading to widenings if these side proofs cannot be closed. Let this syntactic widening be:

$$widen_{fs,n,m,k} : \mathsf{Trm}_{LocSet} \to \mathsf{Trm}_{LocSet}$$

Example 6. *Continuing Example 5, we can calculate $widen_{fs,n,m,k}(singleton(l, \texttt{value}))$. For this we open side proofs to try and prove:*

1. *All object fields in $singleton(l, \texttt{value})$ are in the set fs, i.e. that fs contains the field* `value`*.*
2. *For each object field in $singleton(l, \texttt{value})$, the number of different objects paired with this field is less than or equal to n, i.e. that $n > 0$.*
3. *For each object in $singleton(l, \texttt{value})$, the number of different array index fields paired with this object is less than or equal to m. This is trivially proven, as there are no array index fields in $singleton(l, \texttt{value})$.*
4. *The number of different objects paired with any array index field in $singleton(l, \texttt{value})$ is less than or equal to k. Again, this is trivially proven, as there are no array index fields in $singleton(l, \texttt{value})$.*

This leads to three separate cases:

fs does not contain the field `value`: *Then side proof 1 is not provable and therefore:*

$$widen_{fs,n,m,k}(singleton(l, \texttt{value})) = allLocs \tag{4.41}$$

fs contains the field `value` and $n = 0$: *Then side proof 2 is not provable and therefore:*

$$widen_{fs,n,m,k}(singleton(l, \texttt{value})) = singleton(l, \texttt{value}) \dot{\cup} allObjects(\texttt{value}) \tag{4.42}$$

Which can be simplified to:

$$allObjects(\texttt{value})$$

fs contains the field `value` and $n > 0$: *Then all side proofs can be proven, so:*

$$widen_{fs,n,m,k}(singleton(l, \texttt{value})) = singleton(l, \texttt{value}) \tag{4.43}$$

Abstracting, joining (and possibly widening) heap terms h and h' can thus be performed by calling $abstract_{old}(h, h', anonHeap)$, with $anonHeap$ a fresh heap term and:

$$
\begin{aligned}
&abstract_{old} : (\mathsf{Trm}_{Heap} \times \mathsf{Trm}_{Heap} \times \mathsf{Trm}_{Heap}) \to \mathsf{Trm}_{Heap}\\
&(h, h', anonHeap) \mapsto \begin{cases} anonHeap & \text{, if } locs_{old}(h, \dot{\emptyset}) = \top \\ & \quad \text{or } locs_{old}(h', \dot{\emptyset}) = \top \\ anon(old, & \\ \qquad widen_{fs,n,m,k}(locs_{old}(h, \dot{\emptyset}) \mathbin{\dot\cup} locs_{old}(h', \dot{\emptyset})), & \\ \qquad anonHeap) & \text{, otherwise} \end{cases}
\end{aligned}
$$

Example 7. *Continuing Example 6, we have:*

$$
abstract_{h_1}(h_1, store(h_1, l, \mathtt{value}, 0)) = \begin{cases} anon(h_1, allLocs, anonHeap) & \textit{, if } (4.41) \\ anon(h_1, allObjects(\mathtt{value}), anonHeap) & \textit{, if } (4.42) \\ anon(h_1, singleton(l, \mathtt{value}), anonHeap) & \textit{, otherwise} \end{cases}
$$

Thus, if (4.43)*, the sequent for the next iteration would be:*

$$
\Longrightarrow \{\mathtt{l} := \gamma_{\{List, Null\},1} \parallel \mathtt{heap} := anon(h_1, singleton(l, \mathtt{value}), anonHeap)\}
$$
$$
[\mathtt{while\ (l\ !=\ null)\ \{\ l.value\ =\ 0;\ l\ =\ l.next;\ \}}]\phi
$$

Abstracting the value for the program variable `heap` suffices as an anonymizing update for all modified values on the heap and therefore we do not need to be concerned with modifier sets as described in [53, 9], instead relying on the generated fixed point update $\mathcal{U}'$ to (i) anonymize all modified local variables and heap locations, and (ii) constrain the values for local variables as well.

In addition to γ-values expressing abstraction of program variable values at the syntactic level, we want to track additional helpful invariants. Often these helpful invariant will be relational invariants, combining two (or more) program variables, or rather their values in certain updates. The addition of these invariants therefore also allows modeling some forms of relational abstract domains. We therefore search for a fixed point not of only an update, but rather a constraint/update pair.

The idea is that when refining a constraint/update pair $(C_1, \mathcal{U}_1)$ with another constraint/update pair $(C_2, \mathcal{U}_2)$, with the goal of finding a fixed point, to allow part of the information lost in the new update $\mathcal{U}' = (C_1, \mathcal{U}_1) \dot{\sqcup} (C_2, \mathcal{U}_2)$ due to abstraction, to be retained in the form of possibly new constraints. Thus further abstraction can take place not only within the update, but also by refining or removing these new constraints. Important is that the introduction, refinement and removal of these constraints does not allow infinite chains where there were none to begin with. Therefore the set of new constraints must be guaranteed to be finite, while refining of constraints must have only a finite maximal number of steps before removal of the constraint and introduction of new constraints is only ever allowed when joining of values introduces a new abstract element in $\mathcal{U}'$. This creates in essence a lexicographical ordering of abstract elements and constraints, where as both parts are guaranteed to terminate (possibly by widening), a fixed point can be found.

We ensure these rules for new constraints are followed, by the introduction of a *placeholder update* $\hat{\mathcal{U}}$ within the new constraints. The purpose of the placeholder update is to be substituted by the relevant updates where proving properties is required.

Definition 34 (Placeholder Update $\hat{\mathcal{U}}$)**.** *The* placeholder update $\hat{\mathcal{U}}$ *is a reserved symbol of type* Upd*. No rules for the application of this update exist. Given an update $\mathcal{U}$, the placeholder update can be replaced in a term, formula or set of constraints with substitution $[\hat{\mathcal{U}}/\mathcal{U}]$.*

We introduce the method *refine*, shown in Algorithm 1 on page 64, allowing a constraint set C, which may contain the placeholder update $\hat{\mathcal{U}}$, and an update $\mathcal{U}_1$ to be refined (with respect to the proof P) by a constraint/update pair $(C_2, \mathcal{U}_2)$. The result is a constraint/update pair $(C', \mathcal{U}')$ expressing a refinement of $(C, \mathcal{U}_1)$, where C' may contain the placeholder update $\hat{\mathcal{U}}$. If *refine*$(C, \mathcal{U}_1, C_2, \mathcal{U}_2)$ returns $(C, \mathcal{U}_1)$, i.e. no refinement took place, this can be seen as a refinement fixed point being found.

Example 8. *Consider the case where C does not contain the placeholder update $\hat{\mathcal{U}}$ and no invariant patterns exist. The call refine$(C, \mathcal{U}_1, C_2, \mathcal{U}_2)$ sets C_1 to C on line 1, as C does not contain $\hat{\mathcal{U}}$. It further calculates the following update join on line 2:*

$$\mathcal{U}' = (C_1, \mathcal{U}_1) \dot{\sqcup} (C_2, \mathcal{U}_2)$$

If $\mathcal{U}_1$ is (P, C_1)-weaker than $\mathcal{U}'$ at line 3, the call returns $(C, \mathcal{U}_1)$ at line 19, as allValid is initialized to true on line 4 and the **foreach***-loop on lines 6-16 does not modify this value, as C does not contain $\hat{\mathcal{U}}$. Otherwise, it returns $(C', \mathcal{U}')$ at line 29, where $C' = C$, as it is initialized so at line 23 (as C does not contain $\hat{\mathcal{U}}$) and the* **foreach***-loop on lines 24-27 does not change this (as there are no invariant patterns).*

Applications of method refine when invariant patterns do exist (and C potentially contains the placeholder update $\hat{\mathcal{U}}$) are shown in Example 9 on page 65.

Rather than joining updates until a fixed point is found, our fixed point search for constraint/update pairs in order to generate valid by construction specifications (see Chapter 7), will use the refinement method *refine*.

When generating a loop invariant from a sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{while}\ \ldots\ \omega]\phi, \Delta$, we start with the constraint/update pair $(C, \mathcal{U})$, where $C = \Gamma \cup !\Delta$. As we can assume the constraints C when analyzing

$\{\mathcal{U}\}[\pi$ `while ...` $\omega]\phi$, the assumption that $\bigwedge C[\hat{\mathcal{U}}/\mathcal{U}]$ is trivially the case, as C does not contain $\hat{\mathcal{U}}$. We refine $(C, \mathcal{U})$ by the constraint/update pairs given by symbolic execution of the loop body leading to open branches re-entering the loop. These refinements continue until a fixed point $(C', \mathcal{U}')$ is found, where the refinement guarantees that $\mathcal{U}'$ is weaker than $\mathcal{U}$ and $C \implies \bigwedge C'[\hat{\mathcal{U}}/\mathcal{U}]$ is valid. Together these ensure that the invariant given by $(C', \mathcal{U}')$ is valid initially. Due to the fact that the constraint/update pair is a fixed point, the loop body is also ensured to preserve the invariant. Therefore we can apply a modified version of the update invariant rule introduced in [12].

When introducing new constraints in the method *refine*, we gather the strongest valid invariants for the arguments, where the invariants are instantiations of *invariant patterns*. These invariant patterns contain the placeholder update $\hat{\mathcal{U}}$ and are otherwise explained in detail in the corresponding section introducing each type of invariant pattern.

```
global : Proof $P$
input  : Constraint set $C$ with update $\mathcal{U}_1$ to be refined by constraint/update pair $(C_2, \mathcal{U}_2)$, where
         $\hat{\mathcal{U}}$ does not appear in $C_2$ and we can assume that $\bigwedge C[\hat{\mathcal{U}}/\mathcal{U}_1]$.
output: A refined constraint/update pair.
$C_1 \leftarrow C[\hat{\mathcal{U}}/\mathcal{U}_1]$;
$\mathcal{U}' \leftarrow (C_1, \mathcal{U}_1) \dot\sqcup (C_2, \mathcal{U}_2)$;
if $\mathcal{U}_1$ is $(P, C_1)$-weaker than $\mathcal{U}'$ then
    $allValid \leftarrow true$;
    $C' \leftarrow \emptyset$;
    foreach $c \in C$ do
        if $c$ contains $\hat{\mathcal{U}}$ and $C_2 \implies c[\hat{\mathcal{U}}/\mathcal{U}_2]$ cannot be proven then
            $allValid \leftarrow false$;
            if $c$ can be refined to $c'$, such that $C_2 \implies c'[\hat{\mathcal{U}}/\mathcal{U}_2]$ is valid then
                $C' \leftarrow C' \cup \{c'\}$            /* $C'$ contains a valid refinement of $c$          */
            end
            /* otherwise, neither $c$ nor any refinement thereof will appear in $C'$    */
        else
            $C' \leftarrow C' \cup \{c\}$
        end
    end
    if $allValid$ then
        /* refinement not needed, return the original constraint/update pair        */
        return $(C, \mathcal{U}_1)$
    end
else
    /* new update is strictly weaker, we can introduce new helpful invariants     */
    $C' \leftarrow C \setminus \{c \in C \mid c \text{ contains } \hat{\mathcal{U}}\}$;
    foreach invariant pattern $p$ do
        $c' \leftarrow$ strongest invariant for $p$, such that $C_1 \implies c'[\hat{\mathcal{U}}/\mathcal{U}_1]$ and $C_2 \implies c'[\hat{\mathcal{U}}/\mathcal{U}_2]$ are valid;
        $C' \leftarrow C' \cup \{c'\}$
    end
end
return $(C', \mathcal{U}')$
```

Algorithm 1: Method *refine* to refine one constraint/update pair through another pair.

An important aspect of the heaps our program calculus rules operate on is that these are well-formed heaps, obeying certain properties [63]. Well-formedness is preserved by *create* as well as by *store*, provided a primitive type, or a created object is stored, while *anon* requires both heap arguments are well-formed. As the fresh heap introduced during introduction of γ-symbols for heaps does not need to be well-formed (as otherwise the $\top$ element could not be expressed), we need a way to keep the well-formedness property for those anonymized heaps which are well-formed.

To this end we introduce the following invariant pattern, where `h` is a program variable of type `heap`:

$$\{\hat{\mathcal{U}}\}\mathit{wellFormed}(\mathtt{h}) \tag{5.1}$$

There are no refinement possibilities for this invariant pattern, leading to its removal if no longer valid.

Example 9 (Invariant for well-formed heaps)**.** *Given the initial sequent:*

$$\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null} \Longrightarrow \{\mathtt{heap} := h\}[\mathtt{while\ (o.i\ <\ 0)\ o.i++;}]\phi$$

After one iteration of the loop, we have the new sequent:

$$\begin{aligned}&\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) < 0 \Longrightarrow \\ &\qquad\qquad \{\mathtt{heap} := \mathit{store}(h, \mathtt{o}, \mathtt{i}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) + 1)\}[\mathtt{while\ (o.i\ <\ 0)\ o.i++;}]\phi\end{aligned}$$

We therefore call refine$(C, \mathcal{U}_1, C_2, \mathcal{U}_2)$, with:

$$\begin{aligned}C &= \{\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}\}\\ \mathcal{U}_1 &= (\mathtt{heap} := h)\\ C_2 &= \{\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) < 0\}\\ \mathcal{U}_2 &= (\mathtt{heap} := \mathit{store}(h, \mathtt{o}, \mathtt{i}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) + 1))\end{aligned}$$

Within the method call, line 1 sets C_1 to $C[\hat{\mathcal{U}}/\mathcal{U}_1]$, which (as C does not contain $\hat{\mathcal{U}}$) gives us:

$$C_1 = C = \{\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}\}$$

Joining $(C_1, \mathcal{U}_1)$ with $(C_2, \mathcal{U}_2)$ on line 2 gives us the new update:

$$\mathcal{U}' = (\mathtt{heap} := \mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{f}), h'))$$

As $\mathcal{U}_1$ is not (P, C_1)-weaker than $\mathcal{U}'$, we initialize C' to C on line 23, as C does not contain $\hat{\mathcal{U}}$. Then on lines 25 and 26 we add to C' an instantiation c' of the invariant pattern (5.1)*, where:*

$$c' = \{\hat{\mathcal{U}}\}\mathit{wellFormed}(\mathtt{heap})$$

We add c', as it is the strongest invariant for pattern (5.1)*, where both sequents $C \Longrightarrow c'[\hat{\mathcal{U}}/\mathcal{U}_1]$ and $C_2 \Longrightarrow c'[\hat{\mathcal{U}}/\mathcal{U}_2]$ can be proven:*

Validity of $C \Longrightarrow c'[\hat{\mathcal{U}}/\mathcal{U}_1]$: *Substituting and applying the update gives the trivially valid sequent:*

$$\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null} \Longrightarrow \mathit{wellFormed}(h)$$

Validity of $C_2 \Longrightarrow c'[\hat{\mathscr{U}}/\mathscr{U}_2]$: *Substituting and applying the update gives the sequent:*

$$\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) < 0 \Longrightarrow \mathit{wellFormed}(\mathit{store}(h, \mathtt{o}, \mathtt{i}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) + 1))$$

As store preserves well-formedness of heaps when storing primitives, this sequent is also valid.

The result of refine$(C, \mathscr{U}_1, C_2, \mathscr{U}_2)$ *is therefore*$(C', \mathscr{U}')$*, with:*

$$C' = C \cup \{c'\} = \{\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \{\hat{\mathscr{U}}\}\mathit{wellFormed}(\mathtt{heap})\}$$

The sequent for the next iteration will be $C'[\hat{\mathscr{U}}/\mathscr{U}'] \Longrightarrow \{\mathscr{U}'\}[\mathtt{while\ (o.i\ <\ 0)\ o.i++;}]\phi$*, or:*

$$\begin{aligned}&\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \mathit{wellFormed}(\mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{f}), h')) \Longrightarrow \\ &\qquad\qquad \{\mathtt{heap} := \mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{f}), h')\}[\mathtt{while\ (o.i\ <\ 0)\ o.i++;}]\phi\end{aligned}$$

Symbolic execution (with heap simplification) results in the constraint/update pair:

$$\begin{aligned}C_3 &= \{\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \mathit{wellFormed}(\mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{f}), h')), \mathit{select}(h', \mathtt{o}, \mathtt{i}) < 0\} \\ \mathscr{U}_3 &= (\mathtt{heap} := \mathit{store}(\mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{f}), h'), \mathtt{o}, \mathtt{i}, \mathit{select}(h', \mathtt{o}, \mathtt{i}) + 1))\end{aligned}$$

Calling refine$(C', \mathscr{U}', C_3, \mathscr{U}_3)$ *causes the joining of updates* $(C'[\hat{\mathscr{U}}/\mathscr{U}'], \mathscr{U}') \mathbin{\dot\sqcup} (C_3, \mathscr{U}_3)$*, resulting in:*

$$\mathscr{U}'' = (\mathtt{heap} := \mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{f}), h''))$$

As $\mathscr{U}'$ *is* $(P, C'[\hat{\mathscr{U}}/\mathscr{U}'])$*-weaker than* $\mathscr{U}''$ *and the sequent* $C_3 \Longrightarrow \{\mathscr{U}_3\}\mathit{wellFormed}(\mathtt{heap})$ *can be proven, the method refine returns the fixed point* $(C', \mathscr{U}')$*.*

Note: Our specification generation will use the constraint set $C'[\hat{\mathscr{U}}/\mathscr{U}']$*.*

5.2 Simple Relational Invariants

Some simple relational invariant patterns are (in)equalities between non-integer variables, and any built-in relation between integer variables, such as $>, \geq, <, \leq$ in addition to $\doteq$ and $\neq$.

$$\{\hat{\mathscr{U}}\}(\mathtt{x}_1 \doteq \mathtt{x}_2) \tag{5.2}$$

$$\{\hat{\mathscr{U}}\}(\mathtt{x}_1 \neq \mathtt{x}_2) \tag{5.3}$$

$$\{\hat{\mathscr{U}}\}\chi_a(\mathtt{x}_1 - \mathtt{x}_2) \tag{5.4}$$

For invariant patterns (5.2) and (5.3) between non-integer terms, we allow no refining of these and so removal or weakening the abstract element for $\mathtt{x}_1$ and/or $\mathtt{x}_2$ are the only possible ascending options.

The invariant pattern (5.4) allows invariants $\mathtt{x}_1\ R\ \mathtt{x}_2$ for integer program variables $\mathtt{x}_1, \mathtt{x}_2$ and an integer relation R to be easily expressed in an abstract domain $\mathscr{A}$ for integers, by finding a suitable value for the integer term $\mathtt{x}_1 - \mathtt{x}_2$ within $\mathscr{A}$. This allows for refinement by ascending within $\mathscr{A}$.

Note: In all examples using an integer abstract domain in this chapter, we use the sign domain for integers, so that a in (5.4) can only be in the set $\{\bot, <, 0, >, \leq, \neq, \geq, \top\}$ as we thus do not have to deal with widening. In general any domain can be used, but if widening operators are required in the domain, they will be required for invariant patterns as well.

Example 10 (Relational Integer Invariant). *Given the initial sequent:*

$$\Longrightarrow \{\mathtt{x} := 5 \parallel \mathtt{y} := 1\}[\mathtt{while\ (x\ >\ y)\ y++;}]\phi$$

After one iteration of the loop, we call: refine$(\emptyset, (\mathtt{x} := 5 \parallel \mathtt{y} := 1), \emptyset, (\mathtt{x} := 5 \parallel \mathtt{y} := 2))$

Joining leads to the update:

$$\mathcal{U}' = (\mathtt{x} := 5 \parallel \mathtt{y} := \gamma_{>,1})$$

As the initial update is not (P, C)*-weaker than* $\mathcal{U}'$*, we introduce* $\{\hat{\mathcal{U}}\}\chi_{>}(\mathtt{x} - \mathtt{y})$*, due to the valid sequents:*

$$\Longrightarrow \{\mathtt{x} := 5 \parallel \mathtt{y} := 1\}\chi_{>}(\mathtt{x} - \mathtt{y}) \qquad \Longrightarrow \{\mathtt{x} := 5 \parallel \mathtt{y} := 2\}\chi_{>}(\mathtt{x} - \mathtt{y})$$

The result of refine is $(C', \mathcal{U}')$*, with:*

$$C' = \{\{\hat{\mathcal{U}}\}\chi_{>}(\mathtt{x} - \mathtt{y})\}$$

Symbolic execution of $C'[\hat{\mathcal{U}}/\mathcal{U}'] \Longrightarrow \{\mathcal{U}'\}[\mathtt{while\ (x\ >\ y)\ y++;}]\phi$ *for the next iteration results in:*

$$5 > \gamma_{>,1} \Longrightarrow \{\mathtt{x} := 5 \parallel \mathtt{y} := \gamma_{>,1} + 1\}[\mathtt{while\ (x\ >\ y)\ y++;}]\phi$$

Calling refine$(C', \mathcal{U}', \{5 > \gamma_{>,1}\}, (\mathtt{x} := 5 \parallel \mathtt{y} := \gamma_{>,1} + 1))$ *leads to the joined update:*

$$\mathcal{U}'' = (\mathtt{x} := 5 \parallel \mathtt{y} := \gamma_{>,2})$$

While $\mathcal{U}'$ *is* $(P, C'[\hat{\mathcal{U}}/\mathcal{U}'])$*-weaker than* $\mathcal{U}''$*, the sequent* $C_2 \Longrightarrow \{\mathcal{U}_2\}\chi_{>}(\mathtt{x} - \mathtt{y})$ *cannot be proven, so a fixed point has not yet been found. The invariant must be refined, and so we replace* $\{\hat{\mathcal{U}}\}\chi_{>}(\mathtt{x} - \mathtt{y})$ *with* $\{\hat{\mathcal{U}}\}\chi_{\geq}(\mathtt{x} - \mathtt{y})$*, as this can be shown:*

Validity of $C_2 \Longrightarrow \{\mathcal{U}_2\}\chi_{>}(\mathtt{x} - \mathtt{y})$: *Simplifying gives the valid sequent:*

$$5 > \gamma_{>,1} \Longrightarrow 5 - (\gamma_{>,1} + 1) \geq 0$$

The constraint/update pair for the next iteration is $(C'', \mathcal{U}'')$*, with:*

$$C'' = \{\{\hat{\mathcal{U}}\}\chi_{\geq}(\mathtt{x} - \mathtt{y})\}$$

The next iteration will reveal $(C'', \mathcal{U}'')$ *is a fixed point, resulting in the specification generation using:*

$$(C''[\hat{\mathcal{U}}/\mathcal{U}''], \mathcal{U}'') = (\{5 \geq \gamma_{>,2}\}, (\mathtt{x} := 5 \parallel \mathtt{y} := \gamma_{>,2}))$$

Only allowing instantiations of the invariant pattern for program variables ensures that there are only a finite number of such invariants. However, it can be quite useful to increase the number of program variables before beginning the first refinement, by adding for each program variable `x` a fresh program variable $\mathtt{x}_{\mathtt{old}}$ containing the value for `x` in the initial update. This allows us to track, for example, that the value for `x` can only be increased within a loop.

5.3 Affine Term Invariants

Another source for helpful invariants are terms which have an affine relationship with a loop iteration. In order to be able to introduce such invariants, when analyzing a loop, we introduce a fresh program variable `it`, which stores the value of the current iteration, and add program statements to the loop to ensure that this is so, translating (5.5) into (5.6) before continuing with the loop analysis.

$$\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{while (}g\texttt{) body}\ \omega]\phi, \Delta \qquad (5.5)$$

$$\Gamma \Longrightarrow \{\mathcal{U} \parallel \texttt{it} := 0\}[\pi\ \texttt{while (}g\texttt{) \{ it++; body \}}\ \omega]\phi, \Delta \qquad (5.6)$$

The affine term invariant pattern then has the form:

$$\{\hat{\mathcal{U}}\}t \doteq (c_1 + \{\hat{\mathcal{U}}\}\texttt{it} * c_2) \qquad (5.7)$$

Intuitively, a term t is affine to a loop iteration if $\{\mathcal{U}\}t$ is equal to c_1 for the initial update $\mathcal{U}$ and in an update $\mathcal{U}_n$ expressing the state after executing the loop body n times $\{\mathcal{U}_n\}t$ is equal to $c_1 + n * c_2$. Iteration affine invariants cannot be refined, but can only be removed if no longer valid.

Example 11 (Iteration Affine Invariant)**.** *Given the program and sequent from Example 10:*

$$\Longrightarrow \{\texttt{x} := 5 \parallel \texttt{y} := 1\}[\texttt{while (x > y) y++;}]\phi$$

Before continuing the analysis, we first add the program variable `it`*:*

$$\Longrightarrow \{\texttt{x} := 5 \parallel \texttt{y} := 1 \parallel \texttt{it} := 0\}[\texttt{while (x > y) \{ it++; y++; \}}]\phi$$

Both terms `x` *and* `y` *are iteration affine and as such the fixed point constraint/update pair is:*

$$C' = \{\{\hat{\mathcal{U}}\}\texttt{x} \doteq 5 + \{\hat{\mathcal{U}}\}\texttt{it} * 0, \{\hat{\mathcal{U}}\}\texttt{y} \doteq 1 + \{\hat{\mathcal{U}}\}\texttt{it} * 1\}$$
$$\mathcal{U}' = (\texttt{x} := 5 \parallel \texttt{y} := \gamma_{>,2} \parallel \texttt{it} := \gamma_{\geq,2})$$

C' can be simplifyied to $\{\{\hat{\mathcal{U}}\}\texttt{x} \doteq 5, \{\hat{\mathcal{U}}\}\texttt{y} \doteq 1 + \{\hat{\mathcal{U}}\}\texttt{it}\}$ and $C'[\hat{\mathcal{U}}/\mathcal{U}']$ is $\{5 \doteq 5, \gamma_{>,2} \doteq 1 + \gamma_{\geq,2}\}$.

We allow the term t to be an integer program variable, but also allow certain other terms. In order to ensure only a finite set of affine term invariants can be introduced, we restrict these other terms to only such that appear inside a call to the method *arr*, i.e. are indices in an array access or modification. This will allow for more useful array invariants, as seen in Section 5.5.

Example 12 (Iteration Affine Invariant for an Array Index)**.** *Given the program P and sequent:*

$$\Longrightarrow \{\texttt{heap} := h \parallel \texttt{x} := v\}[\texttt{while (x > 1) \{ x++; a[x] = 0; x=x-3; \}}]\phi$$

Before continuing the analysis, we first add the program variable `it`*:*

$$\Longrightarrow \{\texttt{heap} := h \parallel \texttt{x} := v \parallel \texttt{it} := 0\}[\texttt{while (x > 1) \{ it++; x++; a[x] = 0; x=x-3; \}}]\phi$$

After symbolic execution of a loop iteration we have the following sequent returning to the loop entry:

$$\Longrightarrow \{\texttt{heap} := \mathit{store}(h, \texttt{a}, \mathit{arr}(v + 1), 0) \parallel \texttt{x} := v - 2 \parallel \texttt{it} := 1\}[\texttt{while} \ldots]\phi$$

Refinement results in the constraint/update pair $(\emptyset, \mathcal{U})$, where:

$$\mathcal{U} = (\texttt{heap} := \mathit{anon}(h, \mathit{singleton}(\texttt{a}, \mathit{arr}(v + 1)), h') \parallel \texttt{x} := \gamma_{\geq,1} \parallel \texttt{it} := \gamma_{\geq,2})$$

Symbolic execution of the next iteration gives the following update:

$$\texttt{heap} := \mathit{store}(\mathit{anon}(h, \mathit{singleton}(\texttt{a}, \mathit{arr}(v + 1)), h'), \texttt{a}, \mathit{arr}(\gamma_{\geq,1} + 1), 0) \parallel \texttt{x} := \gamma_{\geq,1} - 2 \parallel \texttt{it} := \gamma_{\geq,2} + 1$$

The term $\gamma_{\geq,1} + 1$ can be found within a call to arr. As we can trace $\gamma_{\geq,1}$ back to the value of `x` *before the iteration, replacing $\gamma_{\geq,1}$ with* `x` *leads to the symbolic pivot (see [34]) $\texttt{x} + 1$. The value for $\texttt{x} + 1$ in the initial update was $v + 1$, while the difference of the values for $\texttt{x} + 1$ between successive updates is -2. All this leads to the generation of the invariant pattern:*

$$\{\hat{\mathcal{U}}\}(\texttt{x} + 1 \doteq v + 1 + \texttt{it} * -2)$$

5.4 Relational Heap-Object Invariants

Relational invariants combining a program variable of type `heap` and a program variable of some subtype of *Object* can allow exploring the fields of an object on the heap. Given program variables `h` of type `heap` and `o` of type *Object*, as well as $\mathtt{f} \in \mathit{Field}$, we can access the value for the field `f` of object `o` on heap `h` by:

$$\mathit{select}(\mathtt{h}, \mathtt{o}, \mathtt{f})$$

Using this, we can generate various invariants regarding this value. First, we can choose an appropriate abstract domain for the type of the field and have the invariant pattern:

$$\chi_a(\mathit{select}(\mathtt{h}, \mathtt{o}, \mathtt{f})) \tag{5.8}$$

Second, we can generate simple relational invariants between the value on the heap and a program variable or a further value on a (possibly different) heap, such as with the invariant patterns:

$$\{\hat{\mathscr{U}}\}(\mathit{select}(\mathtt{h}, \mathtt{o}, \mathtt{f}) \neq \mathtt{x}) \tag{5.9}$$

$$\{\hat{\mathscr{U}}\}\chi_a(\mathit{select}(\mathtt{h}, \mathtt{o}, \mathtt{f}) - \mathtt{x}) \tag{5.10}$$

$$\{\hat{\mathscr{U}}\}(\mathit{select}(\mathtt{h}, \mathtt{o}, \mathtt{f}) \doteq \mathit{select}(\mathtt{h'}, \mathtt{o'}, \mathtt{f'})) \tag{5.11}$$

Third, we can generate affine term invariants using the value on the heap in place of a program variable, with the invariant pattern:

$$\{\hat{\mathscr{U}}\}\mathit{select}(\mathtt{h}, \mathtt{o}, \mathtt{f}) \doteq c_1 + \{\hat{\mathscr{U}}\}\mathtt{it} * c_2 \tag{5.12}$$

In all three cases we only allow field `f` to be an element of the finite set of fields defined by the java program without array index fields, or to be an array index field mentioned explicitly in the location set of a heap γ-symbol, such that we can be ensured of a finite set of invariants which can be so instantiated. Widening in the heap abstraction ensures that only a finite number of array fields will be mentioned explicitly.

Example 13 (Integer abstraction for value on heap)**.** *Given a program P and the sequent:*

$$C \Longrightarrow \{\mathtt{heap} := h\}[\mathtt{while\ (o.i\ <\ 0)\ o.i++;}]\phi$$

Where $C = \{\mathit{wellFormed}(h), \mathtt{o} \neq \mathit{null}, \mathit{select}(h, \mathtt{o}, \mathtt{i}) < 0\}$. Refining after the first iteration introduces not only $\{\hat{\mathscr{U}}\}\mathit{wellFormed}(\mathtt{heap})$, but also the invariant:

$$\{\hat{\mathscr{U}}\}\chi_{\leq}(\mathit{select}(\mathtt{heap}, \mathtt{o}, \mathtt{i}))$$

The constraint set $C'[\hat{\mathscr{U}}/\mathscr{U}']$, where $(C', \mathscr{U}')$ is the fixed point for the loop, is:

$$C \cup \{\mathit{wellFormed}(\mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{i}), h'')), \mathit{select}(\mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{i}), h''), \mathtt{o}, \mathtt{i}) \leq 0\}$$

Which can be simplified to:

$$C \cup \{\mathit{wellFormed}(\mathit{anon}(h, \mathit{singleton}(\mathtt{o}, \mathtt{i}), h'')), \mathit{select}(h'', \mathtt{o}, \mathtt{i}) \leq 0\}$$

If widening in the abstract domain for heaps has resulted in all array index fields for an object of an array type being anonymized, we cannot generate invariants for each of these fields separately, as this would be an infinite set of constraints. We need a way to express invariants about partitions of the array.

The idea is to find an (ideally minimal) overapproximation of all modified fields and create an invariant pattern which contains both an invariant about this partition, as well as the invariant that nothing has changed outside of this partition.

In essence we take a relational heap-object invariant pattern and apply a universal quantifier for a certain partition $P \subseteq \mathbb{Z}$ of array indices on it. At the same time we constrain the array at all indices not in P to be equal to their corresponding value in a different heap. This leads to the general invariant pattern:

$$\{\hat{\mathcal{U}}\}((\forall \texttt{int}\ i; i \notin P; \mathit{select}(\texttt{h}, \texttt{o}, \mathit{arr}(i)) \doteq \mathit{select}(\texttt{h'}, \texttt{o}, \mathit{arr}(i))) \wedge (\forall \texttt{int}\ i; i \in P; \mathit{pattern})) \tag{5.13}$$

Where *pattern* can be, for example:

$$\begin{aligned} &\chi_a(\mathit{select}(\texttt{h}, \texttt{o}, \mathit{arr}(i))) \\ \text{or}\quad &\mathit{select}(\texttt{h}, \texttt{o}, \mathit{arr}(i)) \doteq \texttt{x} \end{aligned}$$

Additionally, as array index fields rely on an underlying integer value, we can allow *pattern* to compare some field $\mathit{arr}(i)$ with some other field $\mathit{arr}(f(i))$ for some function $f : \mathbb{Z} \to \mathbb{Z}$, as in:

$$\begin{aligned} &\mathit{select}(\texttt{h}, \texttt{o}, \mathit{arr}(i)) \doteq \mathit{select}(\texttt{h'}, \texttt{o'}, \mathit{arr}(f(i))) \\ \text{or}\quad &\chi_a(\mathit{select}(\texttt{h}, \texttt{o}, \mathit{arr}(i)) - \mathit{select}(\texttt{h'}, \texttt{o'}, \mathit{arr}(f(i)))) \end{aligned}$$

We must restrict the type of functions f allowed, so as to ensure only a finite number of invariants can be instantiated. This is accomplished by allowing only those functions which appear within *arr* in the update used for refinement.

Example 14 (Array Reversal)**.** *Given a program copying an array* `b` *backwards into another array* `a`*, we consider the following updates encountered during analysis (where $\mathcal{U}$ is the state before and $\mathcal{U}'$ after an iteration of the loop):*

$$\begin{aligned} \mathcal{U} &= (\ldots \parallel \texttt{x} := \gamma_{\geq,1} \parallel \texttt{heap} := h') \\ \mathcal{U}' &= (\ldots \parallel \texttt{x} := \gamma_{\geq,1} + 1 \parallel \texttt{heap} := \mathit{store}(h', \texttt{a}, \mathit{arr}(\gamma_{\geq,1}), \mathit{select}(h', \texttt{b}, \mathit{arr}(\mathit{length}(\texttt{a}) - 1 - \gamma_{\geq,1})))) \end{aligned}$$

Inside arr is the index $\mathit{length}(\texttt{a}) - 1 - \gamma_{\geq,1}$. We can trace $\gamma_{\geq,1}$ back to a program variable, therefore we allow the function $f(i) = \mathit{length}(\texttt{a}) - 1 - i$.

As pattern the following will be quite useful:

$$\mathit{select}(\texttt{heap}, \texttt{a}, \mathit{arr}(i)) \doteq \mathit{select}(\texttt{heap}, \texttt{b}, \mathit{arr}(\mathit{length}(\texttt{a}) - 1 - i))$$

Refinement of invariants using the array invariant pattern (5.13) is possible either by refinement within *pattern*, or by refinement of the chosen partition (or both).

We turn now to what form of partitions are possible, and how these can be refined. At the highest abstraction level before removal of the invariant pattern is the partition containing all legal array indices of the array in `o`. This partition allows reformulating $i \in P$ and $i \notin P$ to:

$$i \in P : \qquad 0 \leq i \wedge i < \mathit{length}(\texttt{o}) \tag{5.14}$$

$$i \notin P : \qquad 0 > i \vee i \geq \mathit{length}(\texttt{o}) \tag{5.15}$$

Beneath this abstraction level are contiguous ranges of indices $[l, r)$, with $0 \leq l \leq r < length(\mathtt{o})$, such that all indices between l (inclusive) and r (exclusive) are within P:

$$i \in P: \qquad l \leq i \land i < r \tag{5.16}$$

$$i \notin P: \qquad l > i \lor i \geq r \tag{5.17}$$

When generating such an invariant, we choose l and r based on the term within *arr* in the refinement update and the value for this term in the initial update.

Example 15 (Contiguous Range Example)**.** *Given the following sequent:*

$$v > 0 \Longrightarrow \{\mathtt{x} := v\}[\mathtt{while\ (x\ <\ b.length)\ \{\ b[x-1]\ =\ 0;\ x\ =\ x*x;\ \}}]\phi$$

Refinement after one iteration gives the update:

$$\mathtt{x} := \gamma_{>,1} \parallel \mathtt{heap} := h'$$

Where h' is the heap abstraction. Symbolic execution from this program state leads to the update:

$$\mathtt{x} := \gamma_{>,1} * \gamma_{>,1} \parallel \mathtt{heap} := store(h', \mathtt{b}, arr(\gamma_{>,1} - 1), 0)$$

The index within arr is $\gamma_{>,1} - 1$. As $\gamma_{>,1}$ can be traced back to the program variable $\mathtt{x}$, we choose $\mathtt{x} - 1$ as upper bound and the value thereof in the initial update ($v - 1$) as lower bound. This gives us an array invariant pattern:

$$\begin{aligned}\{\hat{\mathscr{U}}\}((\forall \mathtt{int}\ i; v - 1 > i \lor i \geq \mathtt{x} - 1; select(\mathtt{heap}, \mathtt{b}, arr(i)) \doteq select(\mathtt{heap_{old}}, \mathtt{b}, arr(i)))\\ \land (\forall \mathtt{int}\ i; v - 1 \leq i \land i < \mathtt{x} - 1; \chi_a(select(\mathtt{heap}, \mathtt{b}, arr(i)))))\end{aligned}$$

While after four iterations, for example, $\mathtt{b}$ has been modified only at indices $\{v - 1, v^2 - 1, v^4 - 1, v^8 - 1\}$, we overapproximate this range with the contiguous range $\{v - 1, v, v + 1, \ldots, v^{16} - 2\}$.

Without any further knowledge of the program, the only abstract element for which this can be proven is $\top$, leading to the simplified invariant:

$$\{\hat{\mathscr{U}}\}\forall \mathtt{int}\ i; v - 1 > i \lor i \geq \mathtt{x} - 1; select(\mathtt{heap}, \mathtt{b}, arr(i)) \doteq select(\mathtt{heap_{old}}, \mathtt{b}, arr(i))$$

Finally, the lowest abstraction level is that of iteration affine ranges. The value c_1 is the first index at which the array will be modified and therefore must be non-negative, as array indices are always non-negative. The value c_2 determines how big the affine step is and in which direction. For an affine range with positive c_2, we have (with % the modulo operator):

$$i \in P: \qquad c_1 \leq i \land i < c_1 + \mathtt{it} * c_2 \land (i - c_1)\%c_2 \doteq 0 \tag{5.18}$$

$$i \notin P: \qquad c_1 > i \lor i \geq c_1 + \mathtt{it} * c_2 \lor (i - c_1)\%c_2 \neq 0 \tag{5.19}$$

While the affine range for a negative c_2 is:

$$i \in P: \qquad c_1 + \mathtt{it} * c_2 < i \land i \leq c_1 \land (c_1 - i)\%(0 - c_2) \doteq 0 \tag{5.20}$$

$$i \notin P: \qquad c_1 + \mathtt{it} * c_2 \geq i \lor i > c_1 \lor (c_1 - i)\%(0 - c_2) \neq 0 \tag{5.21}$$

Note: If $c_2 = 0$, only one value ever gets modified so we need not introduce a range invariant.

Example 16 (Affine Range Example)**.** *Continuing from Example 12, where we had the initial extended sequent:*

$$\Longrightarrow \{\texttt{heap} := h \parallel \texttt{x} := v \parallel \texttt{it} := 0\}[\texttt{while (x > 1) \{ it++; x++; a[x] = 0; x=x-3; \}}]\phi$$

At the start of a later iteration the update was:

$$\texttt{heap} := \mathit{anon}(h, \mathit{singleton}(\texttt{a}, \mathit{arr}(v+1)), h') \parallel \texttt{x} := \gamma_{\geq,1} \parallel \texttt{it} := \gamma_{\geq,2}$$

While the update after symbolic execution of the loop body was:

$$\texttt{heap} := \mathit{store}(\mathit{anon}(h, \mathit{singleton}(\texttt{a}, \mathit{arr}(v+1)), h'), \texttt{a}, \mathit{arr}(\gamma_{\geq,1}+1), 0) \parallel \texttt{x} := \gamma_{\geq,1} - 2 \parallel \texttt{it} := \gamma_{\geq,2} + 1$$

This leads to the affine term invariant:

$$\{\hat{\mathscr{U}}\}(\texttt{x} + 1 \doteq v + 1 + \texttt{it} * -2) \tag{5.22}$$

We can now add the appropriate affine range invariant, where we know each value at a modified index is now 0:

$$\begin{aligned}\{\hat{\mathscr{U}}\}((\forall \texttt{int}\ i; v + 1 + \texttt{it} * -2 \geq i \vee i > v + 1 \vee (v + 1 - i)\%2;& \\ \mathit{select}(\texttt{heap}, \texttt{a}, \mathit{arr}(i)) \doteq \mathit{select}(\texttt{heap}_{\texttt{old}}, \texttt{b}, \mathit{arr}(i)))& \\ \wedge(\forall \texttt{int}\ i; v + 1 + \texttt{it} * -2 < i \wedge i \leq v + 1 \wedge (v + 1 - i)\%2 \doteq 0; \chi_{zero}(\mathit{select}(\texttt{heap}, \texttt{a}, \mathit{arr}(i)))))&\end{aligned} \tag{5.23}$$

Here we lose no precision, as after four iterations, for example, both the actual set of indices at which the array has been modified and our affine range modification set are $\{v + 1, v - 1, v - 3, v - 5\}$.

As can be seen in Example 16, we require that both the affine range invariant (5.23) and the affine term invariant (5.22) can be proven in further iterations. If this is the case, as it is in the example, the set of indices at which the array `a` can be modified is kept very precise. However, without the affine term invariant we cannot prove that the affine range invariant is preserved for any iteration, making this an integral part of the whole.

Note on array aliasing problem

An inherent problem when dealing with arrays is the concept of *array aliasing*, i.e. that multiple program variables of array type contain the same array object. This causes reasoning about an array `a` to have to take into account not only what may happen to `a` itself, but also any other array `b` which might just be the same object as `a`.

As our approach utilizes a theorem prover with a sound calculus for Java to check the validity of potential invariants, we are not susceptible to these problems. Any assumptions made about program variables of array type must still be shown to be valid sequents and therefore aliasing problems will at most cause some invariants to be refined and/or removed, where a better option may have been to modify them slightly.

6 Analysis of Loops With Non-standard Control Flows

The approach to loop analysis given in [12] is based on a fragment of Java not containing any sort of non-standard control flow out of loops. Java, however, has many such possibilities. Ignoring these allows a simple loop unrolling prior to analysis of the loop body in a side proof. Taking non-standard control flows out of loops into account requires re-thinking this.

In this chapter we first show how the JavaDL calculus rules for loops deal with non-standard control flows. We then discuss what our purposes for analyzing loops are and how best to solve these needs, describing the changes necessary for our side proof calculus.

6.1 Unrolling Loops With Non-standard Control Flows

Beckert, Hähnle and Schmitt [9] describe the JavaDL rule for loop unrolling (page 126):

> 'In the general case where break and/or continue occur, the following more complex rule version has to be used:
>
> $$\mathsf{loopUnwind}\ \frac{\Gamma \Longrightarrow [\pi\ \texttt{if (}e\texttt{) l':\{ l":\{ }p'\texttt{ \} }l_1 : \dots l_n : \texttt{while (}e\texttt{) \{ }p\texttt{ \} \}}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow [\pi\ l_1 : \dots l_n : \texttt{while (}e\texttt{) \{ }p\texttt{ \}}\ \omega]\phi, \Delta}$$
>
> where
>
> - l' and l'' are new labels,
> - p' is the result of (simultaneously) replacing in p
>   - every "`break` l_i" (for $1 \leq i \leq n$) and every "`break`" (with no label) that has the `while` loop as its target by `break` l', and
>   - every "`continue` l_i" (for $1 \leq i \leq n$) and every "`continue`" (with no label) that has the `while` loop as its target by `break` l''.
>
>   (The target of a `break` or `continue` statement with no label is the loop that immediately encloses it.)'

The update application $\{\mathscr{U}\}$ is implicitly present before the box modalities of the given rule. Beckert, Hähnle and Schmitt [9] further note (page 127):

> 'In the "unwound" instance p' of the loop body p, the label l' is the new target for `break` statements and l'' is the new target for `continue` statements, which both had the `while` loop as target before. This results in the desired behaviour: `break` abruptly terminates the whole loop, while `continue` abruptly terminates the current instance of the loop body.
>
> A `continue` (with or without label) is never handled directly by a `JAVA CARD` DL rule, because it can only occur in loops, where it is always transformed into a `break` statement by the loop rules.'

Indeed, the `continue` (with or without label) is always transformed into a labeled `break` statement and unlabeled `break` statements are also never handled directly by any JavaDL rule, as they may only occur in loops and switch blocks, where all existing rules transform them into labeled `break` statements.

Note:

Beckert, Hähnle and Schmitt [9] supply a rule for dealing with unlabeled `break` statements, but this rule can never be applied due to the transformations performed, nor would the rule's application make any sense. The rule is not present in the KeY system and we therefore stand by the above statement that unlabeled `break` statements are never handled directly by any JavaDL rule.

6.2 Proving Loop Invariants for Loops With Non-standard Control Flows

The following two approaches for a loop invariant rule for loops with non-standard control flows are given in Beckert, Hähnle and Schmitt [9] on page 148, where the first more theoretical approach is described in detail, while the second more pragmatic approach is the one implemented in KeY:

> Firstly, the logic Java Card DL could be enriched with additional labelled modalities $[]_R$ and $\langle\rangle_R$ with $R \subseteq \{\mathit{break}, \mathit{exception}, \mathit{continue}, \mathit{return}\}$ referring to the reason R of a possible abrupt termination. The semantics of a formula $[\mathtt{p}]_R\phi$ is that, if the program p terminates abruptly with reason R, then the formula ϕ has to hold in the final state, whereas $\langle\mathtt{p}\rangle_R\phi$ expresses that p terminates abruptly with reason R and in the final state ϕ holds.
>
> The second possibility for distinguishing non-termination and abrupt termination is to perform a program transformation such that the resulting program catches all top-level exceptions and thus always terminates normally.’

Both approaches are inherently incomplete. Additionally, a bug in KeY made the implementation unsound.

6.2.1 Theoretical Approach

We consider the rule loopInvariantRule taken from Beckert, Hähnle and Schmitt [9] utilizing labeled modalities for applying a loop invariant, where $AT = \{\mathit{break}, \mathit{exception}, \mathit{return}\}$, $AT' = \{\mathit{exception}\}$ and $\mathscr{U}' = \mathscr{V}(\mathit{Mod})$ with Mod a correct modifier set [53]:

loopInvariantRule

$$\frac{\begin{array}{l}
\Gamma \Longrightarrow \{\mathscr{U}\}\mathit{Inv}, \Delta \\
\Gamma, \{\mathscr{U}'\}\mathit{Inv} \Longrightarrow \{\mathscr{U}'\}[\mathtt{boolean\ v=}\mathit{nse}\mathtt{;}](\mathtt{v} \doteq \mathrm{TRUE} \rightarrow ([\mathtt{p}]\mathit{Inv} \ \wedge\ [\mathtt{p}]_{\mathit{continue}}\mathit{Inv})), \Delta \\
\Gamma, \{\mathscr{U}'\}\mathit{Inv}, \{\mathscr{U}'\}\langle\mathtt{boolean\ v=}\mathit{nse}\mathtt{;}\rangle_{AT'}\ \mathrm{true} \Longrightarrow \{\mathscr{U}'\}[\pi\ \mathtt{v=}\mathit{nse}\mathtt{;}\ \omega]\phi, \Delta \\
\Gamma, \{\mathscr{U}'\}\mathit{Inv}, \{\mathscr{U}'\}\langle\mathtt{boolean\ v=}\mathit{nse}\mathtt{;}\rangle(\mathtt{v} \doteq \mathrm{TRUE} \ \wedge\ \langle\mathtt{p}\rangle_{AT}\ \mathrm{true}) \Longrightarrow \{\mathscr{U}'\}[\pi\ \mathtt{v=}\mathit{nse}\mathtt{;}\ \mathtt{p}\ \omega]\phi, \Delta \\
\Gamma, \{\mathscr{U}'\}\mathit{Inv} \Longrightarrow \{\mathscr{U}'\}[\mathtt{boolean}\ \mathit{v}\mathtt{=}\mathit{nse}\mathtt{;}](\mathtt{v} \doteq \mathrm{FALSE} \rightarrow [\pi\ \ \omega]\phi), \Delta
\end{array}}{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \mathtt{while\ (}\mathit{nse}\mathtt{)\ \{\ p\ \}}\ \omega]\phi, \Delta}$$

By removing the inactive statements and remaining program surrounding the loop, it is ensured that control flow will not leave the loop during analysis of the formula $[\mathtt{p}]\mathit{Inv}$, for example. There are, however, a few problems with this approach, starting with the reason given in [9] when they mention why this is not the implemented solution: Introducing so many different modalities means quite an increase in the number of calculus rules which must be introduced into the KeY system, which in addition to involving intensive work also carries the increased risk of introducing errors into the system.

In addition to the problem recognized in [9], by removing the method-frame information contained in π when trying to analyze the loop body p, information is lost without which certain calculus rules can no longer by applied.

Consider the following sequent:

$$\mathtt{obj} \doteq o \Longrightarrow [\mathtt{method\text{-}frame(source=...,this=o):\{while(i\ >\ 0)\{i\ =\ this.x\ -\ obj.x;\}\}}]\mathtt{i} \doteq 0$$

Intuitively it is clear that applying the loop invariant $\mathtt{i} \geq 0$ should suffice to prove this sequent, as the statement `i = this.x - obj.x;` is equivalent to `i = 0;` within the given context and method frame where both `obj` and `this` points to o (which cannot be `null` as `this` references are never `null`). However, applying this loop invariant leads to the following sequents expressing that the body preserves the loop invariant needing to be proven:

$$\mathtt{obj} \doteq o, c > 0 \Longrightarrow \{\mathtt{i} := c \parallel \mathtt{v} := \mathrm{TRUE}\}[\mathtt{i\ =\ this.x\ -\ obj.x;}]i \geq 0 \tag{6.1}$$

$$\mathtt{obj} \doteq o, c > 0 \Longrightarrow \{\mathtt{i} := c \parallel \mathtt{v} := \mathrm{TRUE}\}[\mathtt{i\ =\ this.x\ -\ obj.x;}]_{\mathit{continue}}i \geq 0 \tag{6.2}$$

These sequents cannot be proven, as the information that `this` points to o has been lost. Additionally, knowing from within which class the implementation came is needed in order to correctly resolve method calls. This inherent incompleteness of the theoretical approach is the main reason why we chose a different solution.

6.2.2 Implemented Approach

In the KeY tool the implemented solution was to transform the loop body by:

1. initializing fresh variables (flags) to track if abrupt termination took place and how,
2. initializing fresh variables to track which exception was thrown or what the return value should be in those cases of abrupt termination,
3. replacing `return` statements with setting the return flag, storing the return value (unless the method is `void`) and breaking out of the loop, and
4. replacing `continue` statements which refer to the loop and `break` statements refering to the loop or a block outside of the loop with setting an appropriate flag and breaking out of the loop. Note that although we do not transform `continue` statements refering to a different loop when performing loop body transformation, we will still never encounter these statements during symbolic execution as both rules to deal with loops transform these statements into `break` statements at some point.
5. As thrown exceptions cannot easily be traced to their origin, due to implicitly thrown exceptions and exceptions leaving method calls, the program transformation included wrapping the transformed loop body in a `try-catch` block which catches all throwables and sets an exception flag, as well as storing the caught throwable for later use.

Example 17 (Loop transformation)**.** *Figure 6.1 shows a loop containing abrupt termination possibilities and its transformed loop body generated according to the above rules.*

Where the classical loop invariant rule differentiates between a *use case* branch and a *body preserves invariant* branch, in the KeY tool the *body preserves invariant* branch must also consider abrupt termination of the loop body and therefore in addition to showing that the loop invariant is preserved in some instances, the use case must be proven in other instances. This was done in the following manner: Let π and ω be the inactive statements surrounding the loop invocation, φ be the use case property to prove and inv be the invariant which should hold in each iteration. Further let Γ' and $\mathcal{U}'$ be the new constraint set and update resulting from executing the modified loop body until (abrupt) termination thereof, if the guard held. Then depending on which flags were set due to a `return`, `break` or `continue` statement or a thrown exception:

No flags set: This is the basic case in which the loop body was executed without any abrupt termination. The loop invariant inv must therefore hold in the new program state. It therefore remains to show that $\Gamma' \Longrightarrow \{\mathcal{U}'\}inv$.

The `return` flag set: It remains to show that $\Gamma' \Longrightarrow \{\mathcal{U}'\}[\pi\ \texttt{return returnExpr;}\ \omega]\varphi$. Or a simpler version, in case of a `return` statement without an argument.

The `break` flag for an unlabeled `break` (or labeled `break` refering to the current loop) set: It remains to show that $\Gamma' \Longrightarrow \{\mathcal{U}'\}[\pi\ \omega]\varphi$, as the `break` statement has been fully resolved by leaving the loop.

A `break` flag for a statement `break` $\texttt{l}_i$`;` is set, where $\texttt{l}_i$ is not a label for the current loop: It remains to show that $\Gamma' \Longrightarrow \{\mathcal{U}'\}[\pi\ \texttt{break}\ \texttt{l}_i\texttt{;}\ \omega]\varphi$, as the `break` statement must exit the block labeled $\texttt{l}_i$ before it is fully resolved.

```
while (guard) {
  if (b1) return r;
  if (b2) continue;
  st
}
```

```
    boolean rtrn#1 = false;
    int returnExpr#1;
    boolean cont#1 = false;
    boolean exc#1 = false;
    Throwable thrownExc#1;
    try {
l1:  {
        if (b1) {
          returnExpr#1 = r;
          rtrn#1 = true;
          break l1;
        }
        if (b2) {
          cont#1 = true;
          break l1;
        }
        st
      }
    }
    catch (Throwable t) {
      exc#1 = true;
      thrownExc#1 = t;
    }
```

Figure 6.1: Transforming Loop Body Containing Abrupt Termination

The `continue` flag set: It remains to show that $\Gamma' \Longrightarrow \{\mathcal{U}'\}inv$, as the `continue` flag marks that control flow would remain inside the loop and therefore requires ensuring that the invariant holds.

The flag due to a caught exception is set: As the exception was caught outside of the loop body, it therefore was thrown within the loop body and not caught or otherwise dealt with. It therefore remains to show that $\Gamma' \Longrightarrow \{\mathcal{U}'\}[\pi$ `throw thrownExc;` $\omega]\varphi$.

There are, however, many problems with this approach. They all result from the same source problem: that the Java language does not easily allow program transformation, as program statements which seem to imply something (for example a `return` statement returning from the method call) only ever *attempt* to perform these actions, but can be stopped from doing so for various reasons: exceptions being thrown while calculating the return value, `finally`-blocks being executed and redirecting control flow, etc. The resulting problems can be summed up as follows:

Multiple flags set: Java control flow only ever allows abrupt termination for a single reason. An abrupt termination for reason X while attempting to terminate abruptly for reason Y will result only in abrupt termination for reason X. The reason Y is completely forgotten. However, the solution implemented in KeY allows multiple flags to be set without it being clear which reason was the last and therefore only relevant reason. This is at the least inherently incomplete, as the only safe solution is to treat multiple flags being set such that we must prove all resulting sequents based on each flag which is set, although it is of course known that only one of these caused the abrupt termination of the loop.

Flag(s) for abrupt termination set in spite of normal termination: This is a much more serious problem than mere incompleteness, as this makes the loop invariant rule unsound. On the positive side it is caused only by code specifically chosen to exploit this weakness, code which would otherwise never occur in real programs. However, it remains a serious problem. To illustrate we examine the loop in

```
while (guard) {
  try {
    try {
      break;
    } finally {
      throw new Exception();
    }
  } catch (Exception e) {
    // ignore
  }
}
```

Listing 6.1: No Abrupt Termination of Loop, Despite Flags Set

Listing 6.1. While it appears at first glance that the loop will be terminated abruptly by the `break` statement, the `finally` block must first be executed. This throws an exception which is caught within the loop and ignored. The result of all this is that the loop is not abruptly terminated at all! However, due to the program transformations on the loop body, the loop invariant rule assumes the loop was abruptly terminated due to an encountered `break` statement and therefore relies on the proof of an irrelevant sequent while ignoring the actual control flow and therefore the relevant sequent.

These problems can be solved by extending the program transformation to `finally` blocks as well, such that the cause of abrupt termination is saved at the beginning of the `finally` block before resetting the flag and if the end of the block is reached the flag for said cause is set once again. This allows abrupt termination from within a `finally` block to in essence overwrite the cause for abrupt termination. However, the fact that program transformation is a tricky subject which allowed the problem in the first place is the motivation to develop a solution with little to no program transformation.

Note that correctly modelling abrupt termination within `finally` blocks has stumped other researchers as well. The tool *Joogie* [5] translates Java sourcecode into bytecode before analyzing it for *infeasible code*, i.e. code for which no feasible execution path exists which leads to its execution. The authors recognize some of the problems with `finally` blocks: as these do not exist in the translated bytecode, the contents of the `finally` block is instead copied for each potential exit point. This can lead to infeasible bytecode in one copy without infeasible Java sourcecode, as the bytecode in another copy might not be infeasible. Due to this or other reasons their implementation fails to correctly identify whether infeasible code exists within examples similar to Listing 6.1, erring on the side of caution by stating that it cannot make any definitive statement about the existence of infeasible code. Huisman and Jacobs [40] describe an approach to Java program verification for code containing abrupt termination, focusing on `break`, `continue` and `return` statements, as well as the corresponding points at which the abrupt termination is "caught" and normal termination restored. They briefly mention abrupt termination via exceptions and the handling thereof with `try-catch`(`-finally`) blocks, but fail to recognize that `finally` blocks must also be considered a catching mechanism for the other forms of abrupt termination as well. However, in [41] Jacobs describes the solution for `try-catch-finally` blocks, which correctly deal also with other forms of abrupt termination.

6.3 Generating Loop Invariants for Loops With Non-standard Control Flows

Our approach to automatic generation of loop invariants does not mesh perfectly with either of the approaches to loops with non-standard control flows, as in both of those cases leaving the loop exceptionally produced additional proof obligations of some sort. While we are generating a loop invariant, i.e. a statement valid at every iteration *entry point*, leaving the loop exceptionally immediately reduces our interest in that branch to zero. We need not consider a single further statement execution, update application or proof simplification on any proof branch which will not reach the next iterations entry point. For this reason our loop body analysis can be simplified in some ways. We do, however, require the addition of new calculus rules for `continue` and unlabeled `break` statements which previously were not required, as these statements could never become the active statement via the existing JavaDL rules.

6.3.1 Unrolling the Loop

The analysis of loops is started when encountering a sequent of the form:

$$\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{while (g) st}\ \omega]\varphi, \Delta \tag{6.3}$$

$$\text{or} \qquad \Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1\texttt{:}\ldots l_n\texttt{:}\ \ \texttt{while (g) st}\ \omega]\varphi, \Delta \tag{6.4}$$

For our side proof, ω does not interest us at all, as it is only relevant when the loop is left, normally or exceptionally. For this reason most of π is also not of interest. However, π can contain method frames, which contain information possibly of relevance while analyzing the loop body, such as class names and what the method's `this` reference points to. We therefore cannot completely ignore π and ω. However, we do not want to allow control flow to enter any part of π or ω, as we are interested only in a single execution of the loop's body, followed by the decision whether we are continuing the loop, or leaving it for any reason. In order to accomplish this, we extend the set of inactive prefix types to include an opening *loop scope* $\circlearrowright$ and also add a matching closing loop scope $\circlearrowleft$. No calculus rules exist which will allow a loop scope to be exited, ensuring that at all times symbolic execution performs rules only inside the loop scope.

Furthermore, we need not perform loop unrolling in the sense that one iteration is unrolled in front of the loop, as in the side proof we do not actually care about the original loop, but only its body. In [12] the original loop is used only as a marker, in that if symbolic execution of the body results in the original loop being reached again, the loop will be continued. We can instead use the statement "`continue;`" as a marker. The sequent which will be given to the side proof, where ψ is an uninterpreted predicate ensuring no updates or constraints will be lost, is therefore:

$$\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \circlearrowright\ \texttt{if (g) \{ st continue; \}}\ \circlearrowleft\ \ \omega]\psi, \Delta \tag{6.5}$$

$$\text{or} \qquad \Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \circlearrowright\ \texttt{if (g)}\ l_1 : \ldots l_n \texttt{:\{ st continue; \}}\ \circlearrowleft\ \ \omega]\psi, \Delta \tag{6.6}$$

As can be seen we need not change the loop body `st`.

6.3.2 Addition of New Rules

Unrolling the loop body verbatim introduces three new active statements which were previously not possible to encounter. There are the unlabeled `break` and both the labeled and unlabeled `continue`. The unlabeled `continue` will also be encountered if the loop body terminates normally, as it has been added after the loop body in the initial sequent given to the side proof. The other approaches turned occurrences of these `break` and `continue` statements within the original loop body into labeled `break` statements. We consider what encountering each of these new statements entails with generation of loop invariants in mind:

`break`: Encountering an unlabeled `break` attempts to exit the loop. If the unlabeled `break` therefore manages to propagate all the way to the loop scope, this signals that it was not ignored via a finally block as in Listing 6.1 and therefore leads to an irrelevant branch. In order to be able to perform this propagation we therefore need to introduce some new rules:

$$\textsf{blockBreakNoLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{break;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1, \ldots, l_n\texttt{:\{ break;}\ p\ \texttt{\}}\ \omega]\varphi, \Delta}$$

$$\textsf{tryBreakNoLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ r\ \texttt{break;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{try\{ break; }p\ \texttt{\}}\ cs\ \texttt{finally\{ }r\ \texttt{\}}\omega]\varphi, \Delta}$$

`continue`: Encountering an unlabeled `continue` attempts to exit the current loop iteration and re-enter the loop. If this is successful, this branch is relevant for the genation of loop invariants. But, as noted above, we must first propogate the `continue` statement upward in case a finally block is used to ignore its control flow. For this propagation we introduce new rules:

$$\textsf{blockContinueNoLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{continue;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1, \ldots, l_n\texttt{:\{ continue; }p\ \texttt{\}}\ \omega]\varphi, \Delta}$$

$$\textsf{tryContinueNoLabel}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ r\ \texttt{continue;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{try\{ continue; }p\ \texttt{\}}\ cs\ \texttt{finally\{ }r\ \texttt{\}}\omega]\varphi, \Delta}$$

`continue` l_i: Encountering a labeled `continue` statement attempts to exit the current iteration *of the loop targeted by the label* and re-enter said loop. This can be the innermost loop, making the labeled `continue` equivalent to an unlabeled `continue`, or it could be a surrounding loop, in which case before exiting that loop's current iteration we first have to exit the innermost loop abruptly, making such a labeled `continue` equivalent to a `break` for our purposes of evaluating whether the branch is relevant or not.

```
outer:  while (se1)
inner:    while (se2) {
            continue inner;
          }
```

```
outer:  while (se1)
inner:    while (se2) {
            continue outer;
          }
```

Figure 6.2: Continuing With the Inner or Outer Loop

In order to understand our reasoning when introducing new rules for labeled continues, consider the two program fragments shown in Figure 6.2. Step by step we now show the sequents reached or generated for side proofs in the analysis of these loops, to easily demonstrate the difference between the two. (The top sequent is always for the inner continuation, the bottom sequent for outer continuation.) Upon first encounter of the outer loop, we start a new side proof:

$$\Longrightarrow [\circlearrowright\ \texttt{if (se1) outer:\{inner: while (se2)\{continue inner;\} continue;\}}\circlearrowleft]\psi$$
$$\Longrightarrow [\circlearrowright\ \texttt{if (se1) outer:\{inner: while (se2)\{continue outer;\} continue;\}}\circlearrowleft]\psi$$

In this side proof we encounter the inner loop:

$$\texttt{se1} \doteq \texttt{true} \Longrightarrow [\circlearrowright\ \texttt{outer:\{inner: while (se2)\{continue inner;\} continue;\}}\circlearrowleft]\psi$$
$$\texttt{se1} \doteq \texttt{true} \Longrightarrow [\circlearrowright\ \texttt{outer:\{inner: while (se2)\{continue outer;\} continue;\}}\circlearrowleft]\psi$$

This leads to a new side proof:

$$\texttt{se1} \doteq \texttt{true} \Longrightarrow [\circlearrowright\ \texttt{outer:\{}\circlearrowright\ \texttt{if (se2) inner:\{\{continue inner;\} continue;\}}\circlearrowleft\ \texttt{continue;\}}\circlearrowleft]\psi$$
$$\texttt{se1} \doteq \texttt{true} \Longrightarrow [\circlearrowright\ \texttt{outer:\{}\circlearrowright\ \texttt{if (se2) inner:\{\{continue outer;\} continue;\}}\circlearrowleft\ \texttt{continue;\}}\circlearrowleft]\psi$$

Simplifying until the labeled `continue` is the active statement gives us:

$$\texttt{se1} \doteq \texttt{true}, \texttt{se2} \doteq \texttt{true} \Longrightarrow [\circlearrowright\ \texttt{outer:\{}\circlearrowright\ \texttt{inner:\{\{continue inner;\} continue;\}}\circlearrowleft\ \texttt{continue;\}}\circlearrowleft]\psi$$

$$\texttt{se1} \doteq \texttt{true}, \texttt{se2} \doteq \texttt{true} \Longrightarrow [\circlearrowright\ \texttt{outer:\{}\circlearrowright\ \texttt{inner:\{\{continue outer;\} continue;\}}\circlearrowleft\ \texttt{continue;\}}\circlearrowleft]\psi$$

Our rules must be able to propagate the labeled `continue` statements out of blocks, but also be able to show that one sequent is relevant, while the other is not. We therefore introduce the following rules for labeled continue statements:

blockContinueNoMatch

$$\frac{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \texttt{continue}\ l';\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ l_1 : \ldots l_n \texttt{:\{ continue}\ l';\ p\ \texttt{\}}\ \omega]\varphi, \Delta}\text{, if } \forall i \in \{1, \ldots, n\}.\ l' \neq l_i$$

$$\text{tryContinueLabel}\ \frac{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ r\ \texttt{continue}\ l';\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \texttt{try\{ continue}\ l';\ p\ \texttt{\}}\ cs\ \texttt{finally\{}\ r\ \texttt{\}}\omega]\varphi, \Delta}$$

$$\text{blockContinueLabel}\ \frac{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \texttt{continue;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ l_1 : \ldots l_i : \ldots l_n \texttt{:\{ continue}\ l_i;\ p\ \texttt{\}}\ \omega]\varphi, \Delta}$$

The rules blockContinueNoMatch and tryContinueLabel merely propagate the labeled continue upwards. The rule blockContinueLabel transforms a labeled `continue` into an unlabeled `continue`, which is propagated upwards. The reason we do this is that due to loop scopes this rule will only ever be applicable in the innermost loop scope. A labeled `continue` matching a block label in the innermost loop scope can only be possible if it is the block corresponding to the innermost loop, as `continue` labels must be labels for loops in the original program and all surrounding unrolled loops are outside of the innermost loop scope. Therefore a labeled `continue` in this case is equivalent to an unlabeled `continue`.

6.3.3 Analysis of Open Branches for Invariant Generation

In addition to closed branches, we must consider the following possibilities for open branches, some of which signal relevance, while others do not. Here we describe the new possible open branches after symbolic execution has terminated and how they should be treated in regard to refining the abstract update and finding a fixed point for our generated loop invariant:

Labeled breaks: A sequent $\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \ \circlearrowright\ \texttt{break}\ l;\ \omega]\varphi, \Delta$ is reachable only if:

1. The label l references a block in π (outside of this loop scope),
2. the statement "`break` l;" was encountered during symbolic execution, and
3. this statement left all enclosing labeled, unlabeled and try blocks.

The `break` was therefore successful in exiting the loop and will not have any influence on the loop invariant: the open branch belonging to this sequent can be safely ignored.

Unlabeled breaks: A sequent $\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \ \circlearrowright\ \texttt{break;}\ \omega]\varphi, \Delta$ signifies that:

1. The statement "`break;`" was encountered during symbolic execution, and
2. this statement left all enclosing labeled, unlabeled and try blocks.

The `break` was therefore successful in exiting the loop and if we were concerned with further control flow, the `break` statement should be removed as it has been resolved upon exiting the loop. For our purposes, however, the only important aspect is that the loop has been exited and the open branch belonging to this sequent is therefore irrelevant for the loop invariant.

Labeled continues: A sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \quad \circlearrowleft$ `continue l';` $\omega]\varphi, \Delta$ signifies that:

1. The label l' references a block in π (outside of this loop scope),
2. the statement "`continue` l';" was encountered,
3. this statement was propagated through the enclosing labeled, unlabeled and try blocks until reaching the block belonging to the innermost loop, and
4. the labeled `continue` left the block as a labeled `continue`, as it did not match any labels of the block belonging to the loop.

This `continue` therefore functions as a `break` out of the current loop, in order to continue a different loop. As the current side proof is only interested in determining the loop invariant for the current loop, however, for our purposes this is treated as any other break: the open branch belonging to the sequent can be safely ignored.

Unlabeled continues: A sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \quad \circlearrowleft$ `continue;` $\omega]\varphi, \Delta$ signifies one of three things:

1. The loop body terminated normally, thereby causing the `continue` statement after the loop body to be encountered and propagated out of the enclosing block; or
2. a) an unlabeled `continue` was encountered, and
   b) this `continue` was propagated through the enclosing labeled, unlabeled and try blocks until reaching the loop scope; or
3. a) a labeled `continue` referencing the innermost loop was encountered,
   b) this `continue` was propagated through the enclosing labeled, unlabeled and try blocks until reaching the block belonging to said innermost loop, and
   c) the rule blockContinueLabel transformed the labeled `continue` into an unlabeled `continue`.

Such a sequent, no matter which of the causes for it, signifies that the loop would be re-entered. Furthermore, the update in front of the modality, which contains the active `continue` statement, expresses the program's state at the point the loop should be re-entered.

Returns: A sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \quad \circlearrowleft$ `return;` $\omega]\varphi, \Delta$ or $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \quad \circlearrowleft$ `return` se`;` $\omega]\varphi, \Delta$ signifies that:

1. The `return` statement was encountered within the method frame containing the loop, and
2. this statement was propagated through the enclosing labeled, unlabeled and try blocks until reaching the loop scope.

The loop has therefore been abruptly terminated by the `return` statement. The open branch containing the sequent is irrelevant for the loop invariant.

Thrown exceptions: A sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \quad \circlearrowleft$ `throw` se`;` $\omega]\varphi, \Delta$ signifies one of the following:

1. Symbolic execution of the loop guard encountered the statement "`throw` se`;`" caused by an implicit exception in the guard statement, such as `1 / 0 == 0`. The loop body was therefore never entered and will not be entered.
2. a) Symbolic execution of the loop guard encountered the statement "`throw` se`;`" caused by an implicit or explicit exception from within a method call contained in the guard, and
   b) this exception was not handled (either by a `catch` or `finally` block), allowing the exception to leave the method frame of the method call within the guard.

   In this case as well the loop body was never entered and will not be entered.
3. a) Symbolic execution of the loop body encountered the statement "`throw` se`;`" (caused either explicitly or implicitly), and

    b) this exception was not handled (either by a `catch` or `finally` block), allowing the exception to propagate up to and out of the loop.

  In this case the loop body was entered, but abrupt termination via thrown exception ensures that it will not be re-entered.

In all three cases the open branch belonging to this sequent has no bearing on the loop invariant.

Empty loop scope: A sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright\ \circlearrowleft\ \ \omega]\varphi, \Delta$ signifies that either:

1. The loop guard was evaluated to `false` and therefore the loop body was never entered and will not be entered so this open branch can be safely ignored, or
2. a) A labeled `break` referencing the innermost loop was encountered,

   b) this labeled `break` was propagated through the enclosing labeled, unlabeled and try blocks until reaching the block belonging to the innermost loop, and

   c) the labeled `break` and matching labeled block were removed by use of the calculus rule blockBreakLabel (provided on page 18).

   Therefore the loop was left abruptly by the labeled `break` statement and this open branch can be ignored.

Note: Due to the `continue` statement added after the loop body, the above are the only two possibilities for reaching an empty loop scope.

Note on Open Branches With Method Calls or Inner Loops as Active Statements

As this analysis takes place when generating specifications, new side proofs will be opened for any encountered method calls or inner loops in order to generate specifications also for these (see Chapter 7). The generated specifications for these will then be applied to the loop or method call responsible for initiating the side proof, in order to resolve the method call or inner loop, and the resulting branch(es) will then be symbolically executed further (again, with any method calls or inner loops dealt with in their own side proofs) until none of the resulting open branches contains a method call or inner loop as its active statement.

Introducing Rules to Close Irrelevant Branches

As we already ignore all closed branches as irrelevant during loop invariant generation, we could simply introduce new calculus rules in the side proofs which close the branches we have determined to be irrelevant, rather than analyze these open branches to determine if they are irrelevant. In fact, this is the safer approach when dealing with a calculus which can be stopped due to maximum step count or similar, as this could give false positives due to a symbolic execution not finishing and a partial result resembling an irrelevant or relevant branch.

We therefore introduce the following rules in order to close all irrelevant branches in side proofs.

Rule for `throw`:

$$\mathsf{closeThrowLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright\ \texttt{throw}\ \ expr;\ \ \omega]\varphi, \Delta}$$

Rules for `return`:

$$\mathsf{closeEmptyReturnLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright\ \texttt{return;}\ \ \omega]\varphi, \Delta}$$

$$\textsf{closeReturnLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \circlearrowright \texttt{return}\ \mathit{expr}\texttt{;}\ \omega]\varphi, \Delta}$$

Note that in the case of an application of closeReturnLoopScope it is ensured by our introduction of the loop scope that the expression *expr* is a simple expression and therefore the loop would indeed be abruptly terminated due to the `return` statement itself. However, even if this were not the case the loop would be abruptly terminated, although the reason for this could also be caused by an exception thrown while trying to evaluate *expr*. For this reason it is therefore safe to allow this rule for any expression, rather than just simple expressions.

Rules for `break`:

$$\textsf{closeUnlabeledBreakLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \circlearrowright \texttt{break;}\ \omega]\varphi, \Delta}$$

$$\textsf{closeLabeledBreakLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \circlearrowright \texttt{break}\ l_i\texttt{;}\ \omega]\varphi, \Delta}$$

Rule for empty loop scope:

$$\textsf{closeEmptyLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \circlearrowright \circlearrowleft \omega]\varphi, \Delta}$$

Note that application of the rule closeEmptyLoopScope can be due to either the loop guard being evaluated to `false` or the loop body being left abruptly due to a labeled `break` referencing the current loop.

Rule for labeled `continue`:

$$\textsf{closeLabeledContinueLoopScope}\ \frac{}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \circlearrowright \texttt{continue}\ l_i\texttt{;}\ \omega]\varphi, \Delta}$$

Finding the Invariant Update Based on Open `continue` Branches

As the full generation of specifications involves generating not only loop invariants but also method contracts for recursive methods, we defer to Chapter 7 for full details. A brief overview for the generation of loop invariants is as follows, where *seq* is a sequent $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{while (guard) body}\ \omega]\phi, \Delta$.

1. Set $seq' := seq$, $(C', \mathcal{U}') := (\Gamma \cup !\Delta, \mathcal{U})$.

2. Unroll the loop in seq' using a loop scope as detailed in Section 6.3.1.

3. Symbolically execute the resulting sequent, performing specification generation and application on any encountered method calls or inner loops.

4. Join $(C', \mathcal{U}')$ with the constraint-update pairs $(\Gamma_i \cup !\Delta_i, \mathcal{U}_i)$ resulting from all open branches with sequents $\Gamma_i \Longrightarrow \{\mathcal{U}_i\}[\pi \circlearrowright \texttt{continue;}\ \omega_i]\psi, \Delta_i$.

5. If $(C', \mathcal{U}')$ was not changed, a fixed point has been found and we can return the constraint/update pair $(C', \mathcal{U}')$. Else set $seq' := C' \Longrightarrow \{\mathcal{U}'\}[\pi\ \texttt{while (guard) body}\ \omega]\psi$ and goto 2.

While generation of an invariant update allows us to ignore control flow leaving the loop, any rule actually applying the invariant update must take these non-standard control flows into account. To this end we introduce an indexed loop scope with calculus rules allowing control flow to leave the loop scope while retaining the information about whether the loop has been left abruptly or if control flow would return to the beginning of the loop. Our indexed loop scope has the form $\circlearrowright_{\mathtt{x}} \cdot \circlearrowleft$ where `x` is a program variable of type `boolean`. The idea is to set `x` upon exiting the loop scope so that `x` is `false` if control flow remains within the loop and `x` is `true` if control flow leaves the loop (due to either abrupt termination or the loop guard being `false`).

We introduce the following calculus rules for the indexed loop scope.

Rule for `throw`:

$$\textsf{throwIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;\ throw}\ se;\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{throw}\ se;\ \mathtt{p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

Rules for `return`:

$$\textsf{emptyReturnIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;\ return;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{return;\ p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

$$\textsf{returnIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;\ return}\ se;\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{return}\ se;\ \mathtt{p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

Rules for `break`:

$$\textsf{labeledBreakIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;\ break}\ l_i;\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{break}\ l_i;\ \mathtt{p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

$$\textsf{unlabeledBreakIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{break;\ p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

Rule for empty loop scope:

$$\textsf{emptyIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;}\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

Note that the rule emptyIndexedLoopScope will be applied on sequents resulting from two different types of control flow: Either the loop was not entered due to the loop guard evaluating to `false`, or the loop was exited abruptly via a labeled `break` statement referencing the current loop. In both of these cases the loop is exited and no further steps need to be applied before continuing with the surrounding program. Furthermore, this result is mirrored in the rule unlabeledBreakIndexedLoopScope, such that a labeled `break` referencing the current loop and an unlabeled `break` result in the same behavior.

Rule for labeled `continue`:

$$\textsf{labeledContinueIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x\ =\ true;\ continue}\ l_i;\ \omega]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{continue}\ l_i;\ \mathtt{p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

Rule for unlabeled `continue`:

$$\textsf{continueIndexedLoopScope}\ \frac{\Gamma \Longrightarrow \{\mathscr{U}\}[\mathtt{x\ =\ false;}]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{continue;\ p}\ \circlearrowleft\ \omega]\varphi, \Delta}$$

Note that continueIndexedLoopScope is the only rule that sets `x` to `false` rather than `true` and also the only rule that completely removes the inactive parts of the sequent, rather than keeping π and ω. This is important in order to allow for the improved invariantUpdate rule (6.7), where:

- $(C', \mathscr{U}')$ are a supplied constraint/update pair,
- `z` is a fresh program variable of type `boolean`,
- $\bar{\mathtt{x}}$ is a duplicate free vector of all program variables assigned to in $\mathscr{U}$ or $\mathscr{U}'$,
- $\bar{c}$ is a vector of the same length as $\bar{\mathtt{x}}$ containing fresh constant symbols,
- *cnt* is a non-trivial, variable-free formula containing a fresh constant symbol $\hat{c}$, such as $\hat{c} \doteq 0$,
- $\bar{\gamma}$ is a vector of all γ-symbols introduced in $\mathscr{U}'$, and
- $\exists\bar{\gamma}.\varphi$ is an abbreviation for $\exists\bar{y}.(\chi_{\bar{a}}(\bar{y}) \land \varphi[\bar{\gamma}/\bar{y}])$, with $\bar{y}$ a vector of the same length as $\bar{\gamma}$ containing fresh logical variables.

$$\textsf{invariantUpdate}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathscr{U}\}\bigwedge C', \Delta \\ \Gamma, \{\mathscr{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathscr{U}'\}\bigwedge C' \Longrightarrow \mathit{cnt} \land \exists\bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \{\mathscr{U}'\}[\pi\ \circlearrowright_{\mathtt{z}}\ \mathtt{if\ (}\mathit{nse}\mathtt{)}\ l_1\mathtt{:}\ \ldots l_n\mathtt{:}\ \mathtt{\{\ p\ continue;\ \}}\ \circlearrowleft\ \omega] \\ \qquad\qquad\qquad ((\mathtt{z} \doteq \mathrm{TRUE} \to \phi) \land (\mathtt{z} \doteq \mathrm{FALSE} \to (!(\mathit{cnt} \land (\bar{\mathtt{x}} \doteq \bar{c}))\ \land\ \bigwedge C'))), \Delta\end{array}}{\Gamma \Longrightarrow \{\mathscr{U}\}[\pi\ l_1 : \ldots l_n :\ \mathtt{while\ (}\mathit{nse}\mathtt{)\ \{\ p\ \}}\ \omega]\phi, \Delta} \tag{6.7}$$

The first premiss of invariantUpdate guarantees that C' is a set of constraints which are initially valid. The second premiss guarantees that $\mathscr{U}'$ is $(P, \Gamma \cup \neg\Delta)$-weaker than $\mathscr{U}$. The third premiss contains the body preserves invariant, the use case and the abrupt termination use case in one. In all cases we may assume the invariant, which is given by $\{\mathscr{U}'\}\bigwedge C'$ in the antecedent as well as by the update $\mathscr{U}'$ before the modality. Additionally we have $\mathit{cnt} \land \exists\bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})$ in the succedent, which we cannot prove by itself, as we have no knowledge of the fresh skolem constant $\hat{c}$ and can therefore not prove *cnt* (and in most cases could not prove $\exists\bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})$, as we have no knowledge of the fresh skolem contants $\bar{c}$, either). Symbolically executing the loop body will lead to `z` either being set to `true` or `false`. If `z` is set to `true` it is either due to the loop guard being evaluated to `false` or the loop being abruptly terminated. These are the use case and abrupt termination use case and we must prove ϕ in each of these. If `z` is set to `false` we must prove that the body preserves the invariant. Part of that is proving $\bigwedge C'$ in the resulting update. The other part is proving that the resulting update could have been expressed by $\mathscr{U}'$. Here the negated formula $!(\mathit{cnt} \land (\bar{\mathtt{x}} \doteq \bar{c}))$ lends itself to being shifted to the antecedent (removing the negation) after the resulting update has been applied. This gives us knowledge about the skolem constants $\bar{c}$ which we can now use to prove $\exists\bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})$, while the formula *cnt* is now trivially proven, as it is assumed. In this way *cnt* serves merely as a safeguard in the case that all $\bar{\mathtt{x}}$ in $\mathscr{U}'$ are assigned $\gamma_{\top}$-symbols, as in that case the formula $\exists\bar{\gamma}.\{\mathscr{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})$ is trivially true and would therefore not require that the symbolic execution of the body leads to the loop being continued.

6.5 Further Uses for Indexed Loop Scopes

The introduction of indexed loop scopes allows us to reconsider the currently implemented solutions to dealing with loops in KeY. As loopInvariantRule needs to be fixed in any case, we begin by considering the following loop invariant rule utilizing indexed loop scopes, where Inv is a supplied loop invariant and $\mathcal{U}' = \{\mathcal{V}(Mod)\}\mathcal{U}$ with Mod a correct modifier set [53]:

loopInvariant

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \{\mathcal{U}\}Inv, \Delta \\ \Gamma, \{\mathcal{U}'\}Inv \Longrightarrow \Delta, \{\mathcal{U}'\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{if}\ \mathtt{(}nse\mathtt{)}\ l_1 : \ldots l_n : \mathtt{\{\ p\ continue;\ \}}\ \circlearrowleft\ \omega]((\mathtt{x} \doteq \mathrm{TRUE} \rightarrow \phi) \land \\ \hfill (\mathtt{x} \doteq \mathrm{FALSE} \rightarrow Inv))\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1 : \ldots l_n : \mathtt{while}\ \mathtt{(}nse\mathtt{)}\ \mathtt{\{\ p\ \}}\ \omega]\phi, \Delta}$$

The first premiss of loopInvariant ensures that Inv is initially valid. The second premiss combines use case, body preserves and abrupt termination use case in one. Assuming the loop invariant holds before some iteration, we must show that if the loop ends before or during this iteration then we can prove the use case ϕ and otherwise we can show that the loop invariant is valid at the end of this iteration.

The rule loopInvariant is in essence a special case of the rule invariantUpdate, where:

- $C' = \{Inv\}$, and
- $\mathcal{U}'$ assigns to all non-heap $\mathtt{x}_i$ a $\gamma_{\top}$-element, and to $\mathtt{heap}$ the heap in $\mathcal{U}$ anonymized based on Mod, such that all these anonymized values are $\gamma_{\top}$-elements.

As $C' = \{Inv\}$ and therefore $\bigwedge C' = Inv$, the first premiss of loopInvariant matches the first premiss of invariantUpdate. The second premiss of invariantUpdate is trivially proven, due to the choice of $\gamma_{\top}$-elements for all changed values between $\mathcal{U}$ and $\mathcal{U}'$. In the third premiss of invariantUpdate $\bigwedge C' = Inv$ as well, and the premiss can further be simplified, as $\exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})$ can be trivially proven in the case of loop continuation, due to the $\gamma_{\top}$-elements and the fact that Mod is a correct modifier set, ensuring that no other values were changed by the loop body.

In addition to being sound, loopInvariant requires no program transformation of the loop body itself and does not require the introduction of multiple modalities.

While the rule for unwinding loops does not need to be changed, we can consider replacing it with the following rule utilizing indexed loop scopes:

loopUnwind

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \Delta, \{\mathcal{U}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \mathtt{if}\ \mathtt{(}nse\mathtt{)}\ l_1 : \ldots l_n : \mathtt{\{\ p\ continue;\ \}}\ \circlearrowleft\ \omega]((\mathtt{x} \doteq \mathrm{TRUE} \rightarrow \phi) \land \\ \hfill (\mathtt{x} \doteq \mathrm{FALSE} \rightarrow [\pi\ l_1 : \ldots l_n : \mathtt{while}\ \mathtt{(}nse\mathtt{)}\ \mathtt{\{\ p\ \}}\ \omega]\phi))\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ l_1 : \ldots l_n : \mathtt{while}\ \mathtt{(}nse\mathtt{)}\ \mathtt{\{\ p\ \}}\ \omega]\phi, \Delta}$$

The advantage of this rule is that it does not require program transformation of the loop body. The disadvantages are that it is a bit harder to understand and the introduction of fresh program variables each time the loop is unrolled is somewhat ugly. However, we offer it as a possible alternative and due to its close similarity to the loopInvariant rule we feel it is worth considering replacing both loop related rules in KeY with these rules based on indexed loop scopes.

7 Specification Generation

In this chapter we demonstrate the entire process for the automatic generation of specifications. We modify the symbolic execution engine during program analysis in the following manner:

1. We add all calculus rules from Chapter 6 related to indexed and unindexed loop scopes, while removing the automatic application of any loop unrolling rules.

2. We remove the automatic application of all calculus rules related to the instantiation and removal of method-frames.

The above allows user defined specifications to be applied in the usual manner to loops and method calls, while otherwise preventing the unrolling of loops and expanding of method bodies. Symbolic execution stopped due to a method call or loop for which no applicable specification exists is used as an entry point for the automatic generation of a specification for this method call or loop. The application of calculus rules allowing method calls to return (normally or exceptionally) is also prevented, enabling the side proofs to gather the information required in order to generate method contracts.

In order to more easily reason about sequents, we introduce the notion of a *normal form* for sequents.

Definition 35 (Normal Form Sequent)**.** *A sequent is in* normal form*, if there exists exactly one formula* $\{\mathcal{U}\}[\mathrm{p}]\phi$ *in the succedent of the sequent. No other formula in the antecedent or succedent may contain a modality.*

Let the method *gatherSequents* take a normal form sequent *seq* as argument and return the set of normal form sequents belonging to open branches of the symbolic execution tree resulting from symbolic execution of the program in the sole modality in *seq*.

The process for analyzing a sequent in order to generate all required specifications for it is as follows: *We repeatedly apply gatherSequents interleaved with the generation and application of specifications for the loops and method calls causing the symbolic execution to stall.*

The restriction to normal form sequents allows the further explanations in this chapter to focus on the relevant points rather than be complicated with special cases for sequents containing multiple formulas with modalities. The restriction itself is not particularly grave, as:

1. The sequents initially passed as input to have specifications automatically generated and applied to them are in normal form.

2. Neither the standard calculus rules, nor the application of our generated specifications produces sequents with additional modalities.

3. If a sequent contains multiple modalities, our approach can still be utilized as follows: First, removing all but one of the formulas containing modalities; then performing the interleaving of *gatherSequents* with application of automatically generated specifications as above; and finally, once the formula containing the modality has been fully handled, adding the removed formulas back into the sequent. This reduces the number of formulas containing modalities in the sequent and by induction this process will be able to automatically generate and apply specifications required for each program in a modality. Further, the information contained in the removed formulas is not lost, as these are added back into the sequent.

In Section 7.1 we show how loop invariants are generated and applied, in addition to discussing how nested loops are handled. Section 7.2 then describes how method contracts are generated and applied, with details on handling mutual recursion and recursive calls within loops. While not illustrated in this chapter, the sound specifications returned by the methods *generateLoopInvariant* and *generateContract* (see Algorithms 2 and 6) can be stored and used to annotate the source code.

When symbolic execution of the input sequent encounters a sequent with a loop as active statement (see (7.1) for the form such a statement has, where *nse* is any expression, *labels* is a possibly empty string of labels $l_1 : \ldots l_n :$ and `body` is any program statement), the method *generateLoopInvariant* (see Algorithm 2) is called with the constraint set $\Gamma \cup !\Delta$, update $\mathscr{U}$ and the program `p` shown in (7.2).

$$\Gamma \Longrightarrow \{\mathscr{U}\}[\pi \ \ labels \ \texttt{while (}nse\texttt{) body} \ \omega]\phi, \Delta \tag{7.1}$$

$$\pi \circlearrowright \texttt{if (}nse\texttt{)} \ \ labels \ \texttt{\{ body continue; \}} \circlearrowleft \omega \tag{7.2}$$

```
input : Initial constraints $C$, update $\mathscr{U}$ and program p (loop body inside a loop scope)
output: A fixed point constraint/update pair expressing the loop invariant
$(C', \mathscr{U}') \leftarrow (C, \mathscr{U})$;
repeat
    $(C'', \mathscr{U}'') \leftarrow (C', \mathscr{U}')$;
    $seq \leftarrow C'[\hat{\mathscr{U}}/\mathscr{U}'] \Longrightarrow \{\mathscr{U}'\}[\mathrm{p}]\psi$;          /* create a sequent from $(C', \mathscr{U}', \mathrm{p})$ */
    /* evaluate $seq$, generating and applying specifications where needed      */
    $E \leftarrow eval(seq)$;
    /* $E$ contains sequents which would re-enter this loop                     */
    foreach $seq' \in E$ do
        /* $seq'$ has the form $\Gamma_1 \Longrightarrow \{\mathscr{U}_1\}[\pi \circlearrowright$ continue; ... $\circlearrowleft \omega]\psi, \Delta_1$          */
        $(C', \mathscr{U}') \leftarrow refine(C', \mathscr{U}', \Gamma_1 \cup !\Delta_1, \mathscr{U}_1)$;       /* refine with state from $seq'$ */
    end
until $(C', \mathscr{U}') = (C'', \mathscr{U}'')$;
return $(C', \mathscr{U}')$
```

Algorithm 2: Method *generateLoopInvariant*

The method *generateLoopInvariant* works as follows: Beginning with the initial constraint/update pair, a new sequent is created and passed to the method *eval*, which we will discuss shortly. The result of *eval* is a set of sequents $\Gamma_i \Longrightarrow \{\mathscr{U}_i\}[\pi \circlearrowright \texttt{continue;} \ \ldots \circlearrowleft \omega]\psi, \Delta_i$, corresponding to the symbolic executions of a loop iteration leading back to the loop entry. We refine the constraint/update pair based on these sequents and if a fixed point has not yet been found, we continue in this manner, creating a new sequent from the refined constraint/update pair and passing this to the method *eval*.

We now turn to the method *eval*. For a simple loop containing no method calls or nested loops, the part of *eval* shown in Algorithm 3 is sufficient, as in the simplest case symbolic execution of the input sequent in *eval* results in a symbolic execution tree containing closed branches for those cases where the loop body was not entered at all or was entered but left exceptionally, and open branches containing sequents which would re-enter the loop. The set of all such sequents is returned.

7.1.1 Applying the Invariants

As we generate the specifications in such a way as to ensure they are sound in regards to the implementation they are generated from, we do not need to apply the full invariantUpdate rule, but rather only produce a sequent expressing the use case administration of this rule. The use case is triggered for both normal and exceptional termination of the loop. The method *applyLoopInvariant* shown in Algorithm 4 demonstrates how the application of a loop invariant in the form of a constraint/update pair is applied to a sequent

```
input  : A sequent seq
output: A set of sequents with active continue statements just inside a loop scope
$L \leftarrow \emptyset$;
$E \leftarrow gatherSequents(seq)$;
while $E \neq \emptyset$ do
    /* Deal with continues in loop scopes                                          */
    if there is a seq $\in E$, where seq has the form: $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi \circlearrowright$ continue; ... $\circlearrowleft \omega]\phi, \Delta$
    then
        $E \leftarrow E \setminus \{seq\}$;                         /* remove sequent from $E$ */
        $L \leftarrow L \cup \{seq\}$;                        /* add sequent to result set */
        continue
    end
    /* Deal with returns (see Algorithm 8)                                         */
    ...
    /* Deal with throws (see Algorithm 9)                                          */
    ...
    /* Deal with method calls (see Algorithm 11)                                   */
    ...
    /* Deal with loops (see Algorithm 5)                                           */
    ...
end
return $L$
```

Algorithm 3: Method *eval*

where the active statement is a loop, producing a resulting sequent with an indexed loop scope expressing the use case for this application.

```
input  : Loop invariant as constraint/update pair $(C', \mathcal{U}')$ and sequent of the form:
         $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi$ labels while (nse) body $\omega]\phi, \Delta$
output: The use case sequent resulting from applying the loop invariant
z $\leftarrow$ a fresh program variable of type boolean;
/* create a sequent from $(C', \mathcal{U}')$, z and the input sequent                 */
return
    $C'[\hat{\mathcal{U}}/\mathcal{U}'] \Longrightarrow \{\mathcal{U}'\}[\pi \circlearrowright_{z}$ if (nse) labels { body continue; } $\circlearrowleft \omega](z \doteq TRUE \rightarrow \phi)$;
```

Algorithm 4: Method *applyLoopInvariant*

7.1.2 Nested Loops

Before dealing with the more complex issue of method calls encountered during symbolic execution, let us remain in the realm of loops by examining what the method *eval* does when encountering an inner loop in a sequent of an open branch. This is shown in Algorithm 5.

In the case that an inner loop is encountered, a new call to *generateLoopInvariant* is made with arguments based on the encountered sequent. This new call to *generateLoopInvariant* will then itself call *eval*, such that these two methods operate in a mutually recursive manner. A loop invariant will be generated for this inner loop (possibly generating further invariants for nested loops inside the inner loop) based on the

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with loops                                                    */
    if there is a seq $\in E$, where seq has the form:
                    $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi$ labels while (nse) body $\omega]\phi, \Delta$
    then
        p $\leftarrow \pi$ $\circlearrowright$ if (nse) labels { body continue; } $\circlearrowleft$ $\omega$;
        $(C', \mathcal{U}') \leftarrow generateLoopInvariant(\Gamma \cup !\Delta, \mathcal{U}, \mathtt{p})$;
        $E \leftarrow E \cup gatherSequents(applyLoopInvariant(C', \mathcal{U}', seq))$;
        continue
    end
    ...
end
...
```

Algorithm 5: The part of method *eval* dealing with loops

program state at which the inner loop was initially encountered. This loop invariant will then be applied to the sequent responsible for its creation and the result of this application will be symbolically executed, with resulting sequents of open branches added to the set of sequents *eval* processes. This ensures that the code after the inner loop is taken into account.

Invariants generated for inner loops may be generated multiple times, as the abstract program state describing the entry points of the inner loop in the first iteration of the outer loop may well need to be abstracted further for a later iteration.

Example 18. *Given the sequent* (7.3)*, the initial call to generateLoopInvariant leads to eval encountering the inner loop at sequent* (7.4)*.*

$$\Longrightarrow \{\mathtt{x} := 0 \parallel \mathtt{y} := v\}[\texttt{while(x < y) \{ while (x > 0) \{ x--; y--; \} x++; \}}]\phi \tag{7.3}$$

$$0 < v \Longrightarrow \{\mathtt{x} := 0 \parallel \mathtt{y} := v\}[\circlearrowright \texttt{ \{ \{ while (x > 0) \{ x--; y--; \} x++; \} continue; \} } \circlearrowleft]\psi \tag{7.4}$$

Calling generateLoopInvariant with the initial constraint/update pair $(\{0 < v\},\ \mathtt{x} := 0 \parallel \mathtt{y} := v)$ *and the program in* (7.5) *based on the sequent* (7.4) *results in the constraint/update pair remaining unabstracted, as the inner loop is not entered in this program state and so the method eval therefore returns an empty set of sequents (as none would re-enter the inner loop that is not entered at all), requiring no refinement of the constraint/update pair and its return as a result.*

$$\circlearrowright \texttt{ \{ \{ } \circlearrowright \texttt{ if (x > 0) \{ \{ x--; y--; \} continue; \} } \circlearrowleft \texttt{ x++; \} continue; \} } \circlearrowleft \tag{7.5}$$

Applying the loop invariant $(\{0 < v\},\ \mathtt{x} := 0 \parallel \mathtt{y} := v)$ *to sequent* (7.4) *gives us the following sequent:*

$$0 < v \Longrightarrow \{\mathtt{x} := 0 \parallel \mathtt{y} := v\}[\circlearrowright \texttt{ \{ \{ } \circlearrowright_{\mathtt{z}} \texttt{ if (x > 0) \{ x--; y--; \} } \circlearrowleft \texttt{ x++; \} continue; \} } \circlearrowleft] \\ (\mathtt{z} \doteq \mathit{TRUE} \rightarrow \psi)$$

Symbolic execution leads to the sequent:

$$0 < v \Longrightarrow \{\mathtt{y} := v \parallel \mathtt{z} := \mathit{TRUE} \parallel \mathtt{x} := 1\}[\circlearrowright \texttt{ continue; } \circlearrowleft](\mathtt{z} \doteq \mathit{TRUE} \rightarrow \psi)$$

The initial call to generateLoopInvariant now takes this sequent (the result of eval) and uses it to refine $(\emptyset,\ \mathtt{x} := 0 \parallel \mathtt{y} := v)$*. Using the sign abstract domain, this results in the new constraint/update pair*

$(\emptyset,\ \mathtt{x} := \gamma_{\geq,1} \parallel \mathtt{y} := \upsilon)$*. As no fixed point for the outer loop has been found yet, eval is called again, encountering the inner loop this time at sequent* (7.6)*.*

$$\gamma_{\geq,1} < \upsilon \Longrightarrow \{\mathtt{x} := \gamma_{\geq,1} \parallel \mathtt{y} := \upsilon\}[\circlearrowright\ \mathtt{\{\ \{\ while\ (x\ >\ 0)\ \{\ x--;\ y--;\ \}\ x++;\ \}\ continue;\ \}}\ \circlearrowleft]\psi \tag{7.6}$$

This time the call to generateLoopInvariant for the inner loop is more interesting, resulting in refinement of $(\{\gamma_{\geq,1} < \upsilon\},\ \mathtt{x} := \gamma_{\geq,1} \parallel \mathtt{y} := \upsilon)$ *with* $(\{\gamma_{\geq,1} < \upsilon\},\ \mathtt{x} := \gamma_{\geq,1} - 1 \parallel \mathtt{y} := \upsilon - 1)$*.*

As the example clearly demonstrates, a fixed point invariant for an inner loop in one iteration of the outer loop can change drastically in the next.

7.2 Generating and Applying Method Contracts

Our specifications for recursive method calls in essence contain two types of program state invariants:

Precondition: (An overapproximation of) all program states at the call sites for this method, containing both the initial call and (at least) all recursive calls reachable from this initial call.

Postconditions: (Overapproximations of) all program states resulting from the method returning (normally or exceptionally), when called from a precondition state.

In [60] we showed how *interprocedural dataflow analysis* [54] can be used in order to generate these pre- and postconditions automatically for method calls of a recursive toy language containing no loops and only the integer type for parameters and return value. A solution idea for mutual recursion is included. In this section we extend this idea to allow:

1. inclusion of all Java types for parameters and return values, including the `void`-return type,
2. the method body to contain loops,
3. the program heap as implicit input parameter and result,
4. instance methods with their additional parameter `this`,
5. exceptional behavior and therefore exceptional postconditions, and
6. a detailed algorithm which can also handle mutual recursion.

7.2.1 Outline

The basic idea is to generate the precondition by applying symbolic execution to the expanded method body and joining all program states at which recursive calls are reached. Additionally, postconditions are generated by joining program states at which the method call returns. We differentiate here between normal behavior and exceptional behavior, by joining program states at which the method is returned exceptionally only with each other and analogously for normal returns. Similar to the generation of loop invariants, we seek fixed points for the precondition and postconditions. The following important distinctions must be made, however:

1. In contrast to loops, where we are interested in the values for explicit program variables and can therefore join the unmodified program states, for methods we are interested in the values of the *formal parameters* of the method. The actual parameters are always fresh program variables created by the calculus rule methodCall, such that joining unmodified program states would accomplish nothing.

We therefore need to modify the precondition program states at method calls before joining, to ensure that the modified updates use matching program variables for the method's formal parameters. Additionally, as the method call cannot access any program variables other than its actual parameters and `heap`, the modified update should remove all assignments to other program variables.

2. The precondition and postconditions are not orthogonal to each other. Obviously, the precondition influences the postconditions, but for non-tail recursive methods the postconditions may also influence the preconditions as well, if a second recursive call takes place after returning from the first recursive call. For these reasons a fixed point is only reached if neither precondition nor postconditions change.

3. In order to join program states for the postconditions, these must first be modified to ensure the following:

   a) The program variable for the return value must always be the same when joining normal postconditions.

   b) The program variable for the thrown value must always be the same when joining exceptional postconditions.

   c) Aside from `heap` and the value returned or thrown, the postcondition may not change any program variables.

4. In contrast to loops, where the joined program state will be used only once in the next iteration of the fixed point algorithm, the postconditions may be applied at multiple points. As postcondition updates contain γ-symbols, which could express different values at each of these application points, we must apply modifed postconditions containing fresh γ-symbols.

In this chapter we use an implementation for calculating the n^{th} element of the Fibonacci sequence shown in Listing 7.1 as a running example to demonstrate many of these issues.

```
class Math {
  static int fib(int n) {
    if (n < 0)
      throw new IllegalArgumentException();
    else if (n == 0)
      return 0;
    else if (n == 1)
      return 1;
    else
      return fib(n-2) + fib(n-1);
  }
}
```

Listing 7.1: A method implementing the Fibonacci sequence

7.2.2 Definitions

As mentioned above, we need to ensure that when joining precondition updates, these assign to the same program variables, based on the method's formal parameters. As the program heap is an implicit input parameter, we also require a program variable to store its initial value, as otherwise this knowledge would be lost if the heap is modified. Similarly, we require designated program variables for the return value and thrown value of a method. For these reasons we introduce *placeholder program variables.*

Definition 36 (Placeholder program variables)**.** *For each method identifier* $\mathtt{mname}(\tau_1 \times \cdots \times \tau_n \to \tau)$@*Class, abbreviated* m*, we define the following* placeholder program variables*, which are unique reserved program variables:*

1. $\mathtt{heap}^m$ *of type Heap,*
2. $\mathtt{p}_i^m$ *of type* τ_i *for all* $i \in \{1, \ldots, n\}$*,*
3. $\mathtt{throw}^m$ *of type Throwable* $\sqsubset$ *Object, and*
4. $\mathtt{return}^m$ *of type* τ*, if* τ *is not the* `void` *return type.*

In order to deal with mutual recursion, we need to know within the context of which other method calls an encountered method call lies. To this end we introduce the concept of a *method frame stack.*

Definition 37 (Method Frame Stack)**.** *A* method frame stack *is a data structure for storing non-duplicate method identifiers, with the following (partial) functions:*

empty: *a constant function returning a method frame stack designated as empty.*

push: *takes a method frame stack and a method identifier and returns a new method frame stack with the top element being the pushed element and the remaining stack being the original stack (or undefined if the method identifier is already on the stack).*

pop: *takes a method frame stack and returns the stack without its top element (or undefined for an empty stack).*

get: *takes a method frame stack and a method identifier and returns undefined if the method identifier is not on the stack. Otherwise it returns all elements from the top of the stack up to the method identifier argument (both inclusive).*

Method frame stacks provide the following axioms:

1. *No method identifiers are on the* **empty** *stack.*
2. $\mathbf{pop}(\mathbf{push}(MFS, m)) = MFS$*, if* $\mathbf{push}(MFS, m)$ *is defined.*
3. $\mathbf{get}(\mathbf{push}(\ldots \mathbf{push}(MFS, m_1) \ldots, m_n), m_1) = \{m_1, \ldots, m_n\}$*, if for all* $1 \leq i \leq n$ *the function calls* $\mathbf{push}(\ldots \mathbf{push}(MFS, m_1) \ldots, m_i)$ *are defined.*

The following is a possible implementation of a method frame stack, modeled as sets of sets of method identifiers:

$$\mathbf{empty} = \emptyset$$

$$\mathbf{push}(MFS, m) = \begin{cases} \{\{m\}\} \cup \{X \cup \{m\} \mid X \in MFS\} & \text{, if } \forall M \in MFS.\ m \notin M \\ \textit{undefined} & \text{, otherwise} \end{cases}$$

$$\mathbf{pop}(MFS) = \begin{cases} \{X \setminus \{m\} \mid X \in MFS\} \setminus \{\emptyset\} & \text{, if there is an } m \text{ such that } \forall M \in MFS.\ m \in M \\ \textit{undefined} & \text{, otherwise} \end{cases}$$

$$\mathbf{get}(MFS, m) = \begin{cases} \textit{undefined} & \text{, if } \forall X \in MFS.\ m \notin X \\ M & \text{, where } m \in M \land M \in MFS \land \forall X \in MFS.\ m \in X \to M \subseteq X \end{cases}$$

While gathering the information required to create sound method contracts for encountered method calls, we begin with only partial information about the precondition and no information about postconditions. During analysis more information becomes available, until a sound method contract can be inferred from the fixed point of our *partial method contract.*

Definition 38 (Partial Method Contract)**.** *A* partial method contract *is a tuple* $(m, init, pre, POST)$*, where* m *is a method identifier,* $init$ *is a boolean value,* pre *is a single constraint/update pair and* $POST$ *is a set of constraint/update pairs.*

Given a method identifier m and a set of partial method contracts S containing no duplicate method identifiers, we define:

$$S(m) := \begin{cases} (m, init, pre, POST) & \text{, if there is a } (m, init, pre, POST) \in S \\ \mathit{undefined} & \text{, otherwise} \end{cases}$$

In order to track the partial method contracts created so far and to join these in the case of mutual recursion, we require a *partial method contract mapping.*

Definition 39 (Partial Method Contract Mapping)**.** *A* partial method contract mapping *is a set of tuples* $(M, S, redo)$*, where* M *is a set of method identifiers,* S *is a set of partial method contracts containing no duplicate method identifiers and* $redo$ *is a boolean value, such that the sets of method identifiers in all tuples are disjoint from one another.*

Given a method identifier m and a partial method contract mapping PMC, we define:

$$PMC(m) := \begin{cases} (M, S, redo) & \text{, if there is a } (M, S, redo) \in PMC \text{ where } m \in M \\ \mathit{undefined} & \text{, otherwise} \end{cases}$$

7.2.3 Generating the Method Contracts

Symbolic execution of a sequent with a method call as active statement, such as the call to `Math.fib` in (7.7), still leads to the application of the rule methodCall (shown on page 18) to initialize local variables for the actual parameters and locate the implementation, as well as further rules on the result thereof. The sequents of open branches after symbolic execution will have *fully specified method calls* as active statements. For example symbolic execution of (7.7) will lead to (7.8).

$$v \leq 1 \Longrightarrow \{\texttt{heap} := h \parallel \texttt{x} := v\}[\pi\ \texttt{f = Math.fib(x);}\ \omega]\phi \tag{7.7}$$

$$v \leq 1 \Longrightarrow \{\texttt{heap} := h \parallel \texttt{x} := v \parallel \texttt{x}_1 := v\}[\pi\ \texttt{x}_0\ \texttt{= Math.fib(}\texttt{x}_1\texttt{)@Math; f =}\ \texttt{x}_0\texttt{;}\ \omega]\phi \tag{7.8}$$

At this point the method *generateContract* (see Algorithm 6) is called with the constraints, update, method identifier and a list of the actual parameter program variables gained from the sequent. In the case of sequent (7.8), we call:

$$\mathit{generateContract}(\{v \leq 1\},\ \texttt{heap} := h \parallel \texttt{x} := v \parallel \texttt{x}_1 := v,\ \texttt{Math.fib(int}\rightarrow\texttt{int)@Math},\ \texttt{x}_1)$$

```
global : A method frame stack $MFS$ and a partial method contract mapping $PMC$.
input  : Constraints $C$, update $\mathcal{U}$, method id $m$ and parameter program variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$.
output: A flag stating whether the generation finished after the initial pass and a set of
        postconditions as constraint/update pairs.
/* add $m$ to method frame stack */
$MFS \leftarrow \mathbf{push}(MFS, m)$
/* $m$ has no known mutual recursion with other method ids */
$M \leftarrow \{m\}$;
/* initialize $\mathcal{U}_1$ with read-only input parameters and modifiable heap */
$\mathcal{U}_1 \leftarrow \mathtt{heap} := \{\mathcal{U}\}\mathtt{heap} \parallel \mathtt{heap}^m := \{\mathcal{U}\}\mathtt{heap} \parallel \mathtt{p}_1^m := \{\mathcal{U}\}\mathtt{x}_1 \parallel \ldots \parallel \mathtt{p}_n^m := \{\mathcal{U}\}\mathtt{x}_n$;
/* set initial partial contract for $m$ */
$S \leftarrow \{(m, \mathit{true}, (C, \mathcal{U}_1), \emptyset)\}$;
repeat
    /* add partial contracts */
    $PMC \leftarrow PMC \cup \{(M, S, \mathit{false})\}$;
    foreach $(m', \_, (C', \mathcal{U}'), \_) \in S$ do
        /* evaluate the expanded sequent based on $m', C', \mathcal{U}'$
           resulting (empty) set of sequents is ignored, but $PMC$ may change */
        $\mathit{eval}(\mathit{createExpandedSequent}(C', \mathcal{U}', m))$
    end
    /* find out if our partial contract has reached a fixed point */
    $(M, S, \mathit{redo}) \leftarrow PMC(m)$;
    /* remove partial contracts (if $\mathit{redo}$, they will be added again) */
    $PMC \leftarrow PMC \setminus \{(M, S, \mathit{redo})\}$
until $\mathit{redo}$ is false;
/* remove $m$ from method frame stack */
$MFS \leftarrow \mathbf{pop}(MFS)$;
/* if mutual recursive calls are still on the method frame stack... */
foreach $m' \in (M \setminus \{m\})$ do
    if $\mathbf{get}(MFS, m')$ is defined then
        /* ...add partial contracts back, so they can be reused by those calls */
        $PMC \leftarrow PMC \cup \{(M, S, \mathit{false})\}$;
    end
end
/* return the postconditions for $m$ */
$(m, \mathit{init}, \_, R) \leftarrow S(m)$;
return $(\mathit{init}, R)$
```

Algorithm 6: Method *generateContract*

The method *generateContract* works as follows:

- In order to handle mutual recursion, the method identifier has to be pushed to the global (initially **empty**) method frame stack *MFS* at the start of *generateContract* and removed at the end. Additionally, we use the set $\{m\}$, rather than simply m, in order to allow multiple method identifiers in the case that these are found to be mutually recursive. Furthermore, in general the partial method contracts for the method under consideration are removed from the partial method contracts mapping before returning the postconditions generated for the initial call. In the case of mutual recursion, the partial method contract mapping must retain partial method contracts for all mutually recursive methods until all specifications generated for these methods have been put to use. For details on this, see Section 7.2.8. For the moment we will focus on simple recursion.

- To ensure that preconditions can be joined, we create an update $\mathcal{U}_1$ which assigns the values of the actual parameters to the placeholder parameters for the method. In addition, as the program heap is always an implicit input parameter to all methods, we assign the value of the heap in $\mathcal{U}_1$ to `heap` and also $\texttt{heap}^m$.

- With the initial partial method contract based on the constraints and the newly created update $\mathcal{U}_1$, we now try to find a fixed point. This is accomplished by adding the partial method contract to the global partial method contract mapping, before creating a sequent from it (see Algorithm 7) and then calling *eval* on the result. Within the method *eval*, the partial method contract mapping may be modified. If no modification takes place, then a fixed point has been found for the partial method contract and we can return the postconditions for this sound contract to be applied to the initial method call. Otherwise, in the next iteration of the fixed point search, we will create a new sequent based on the modified partial method contract, continuing until a fixed point is found.

```
input  : Constraints $C$, update $\mathcal{U}$ and method id $m$ = m($sig$)@$Class$
output: The expanded sequent of the form $C' \Longrightarrow \{\mathcal{U}'\}$[method-frame(...){ body }]$\psi$
($\texttt{x}_1, \dots, \texttt{x}_n$) ← fresh program variables of matching type to $\texttt{p}_1^m, \dots, \texttt{p}_n^m$;
$args$ ← $\texttt{x}_1, \dots, \texttt{x}_n$;
$C' \leftarrow C[\hat{\mathcal{U}}/\mathcal{U}]$;
$\mathcal{U}' \leftarrow \mathcal{U} \parallel \texttt{x}_1 := \{\mathcal{U}\}\texttt{p}_1^m \parallel \dots \parallel \texttt{x}_n := \{\mathcal{U}\}\texttt{p}_n^m$;
/* gather the method frame info                                        */
$info$ ← source=$m$;
if $m$ is an instance method then
    /* the receiving object is stored in the first argument            */
    $info$ ← $info$,this=$\texttt{x}_1$;
    $args$ ← $\texttt{x}_2, \dots, \texttt{x}_n$
end
if $m$ has a non-void return type then
    $info$ ← result->$\texttt{return}^m$, $info$;
end
$body$ ← the body for the implementation of $m$ with the formal parameters replaced by $args$
return $C' \Longrightarrow \{\mathcal{U}'\}$[method-frame($info$):{ $body$ }]$\psi$
```

Algorithm 7: Method *createExpandedSequent*

Example 19. *Consider the call:*

$$\mathit{generateContract}(\{v \le 1\},\ \texttt{heap} := h \parallel \texttt{x} := v \parallel \texttt{x}_1 := v,\ \texttt{Math.fib(int}\rightarrow\texttt{int)@Math},\ \texttt{x}_1)$$

The first iteration of the loop will add the following to PMC, where m is `Math.fib(int→int)@Math`*:*

$$(\{m\}, \{(m, \mathit{true}, (\{v \le 1\},\ \texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v), \emptyset)\}, \mathit{false})$$

This leads to the call createExpandedSequent($\{v \le 1\}$, $\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v$, m), resulting in the following sequent being passed to eval:

$$\begin{array}{l}
v \le 1 \Longrightarrow \{\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{x}_1 := v\} \\
\qquad [\texttt{method-frame(result->return}^m\texttt{,source=}m\texttt{):\{} \\
\qquad\quad \texttt{if (x}_1\texttt{ < 0)} \\
\qquad\qquad \texttt{throw new IllegalArgumentException();} \\
\qquad\quad \texttt{else if (x}_1\texttt{ == 0)} \\
\qquad\qquad \texttt{return 0;} \\
\qquad\quad \texttt{else if (x}_1\texttt{ == 1)} \\
\qquad\qquad \texttt{return 1;} \\
\qquad\quad \texttt{else} \\
\qquad\qquad \texttt{return fib(x}_1\texttt{-2) + fib(x}_1\texttt{-1);} \\
\qquad \texttt{\}}]\psi
\end{array}$$

The first step in eval is to call gatherSequents on the input sequent. This results in the set E containing the following three sequents, where h' is the modified heap containing the newly created exception e, while Γ contains additional information regarding h' and e:

$$\begin{array}{l} v \doteq 0 \Longrightarrow \{\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{x}_1 := v \parallel \texttt{return}^m := 0\} \\ \qquad [\texttt{method-frame(source=}m\texttt{):\{ \}}]\psi \end{array} \tag{7.9}$$

$$\begin{array}{l} v \doteq 1 \Longrightarrow \{\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{x}_1 := v \parallel \texttt{return}^m := 1\} \\ \qquad [\texttt{method-frame(source=}m\texttt{):\{ \}}]\psi \end{array} \tag{7.10}$$

$$\begin{array}{l} \Gamma, v < 0 \Longrightarrow \{\texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{x}_1 := v \parallel \texttt{heap} := h' \parallel \texttt{exc} := e\} \\ \qquad [\texttt{method-frame(result->return}^m\texttt{,source=}m\texttt{):\{ throw exc; \}}]\psi \end{array} \tag{7.11}$$

In Section 7.1 we already discussed how *eval* deals with loop re-entry and inner loops. In Algorithm 8 we show how returning from a method normally is dealt with.

In the case that no recursive calls have been encountered for the method, the partial method contract mapping is simply updated with a further postcondition based on the sequent with an empty method-frame. All constraints are kept, but the update is reduced to containing only unmodified input program variables and the output program variables. Local variables introduced in the method call therefore do not leak into the postcondition update.

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with returns */
    if there is a $seq \in E$, where $seq$ has the form:
        $\Gamma \Longrightarrow \{\mathscr{U}\}[\pi$ method-frame(source=$m$,...) { } $\omega]\phi, \Delta$
    then
        $E \leftarrow E \setminus \{seq\}$;    /* remove the sequent from $E$ */
        $(M, S, redo) \leftarrow PMC(m)$;    /* $PMC(m)$ will return a defined value */
        $(m, init, (C', \mathscr{U}'), R) \leftarrow S(m)$;
        /* create normal postcondition update with preconditions... */
        $\mathscr{U}_R \leftarrow \mathtt{heap}^m := \{\mathscr{U}\}\mathtt{heap}^m \parallel \mathtt{p}_1^m := \{\mathscr{U}\}\mathtt{p}_1^m \parallel \ldots \parallel \mathtt{p}_n^m := \{\mathscr{U}\}\mathtt{p}_n^m$;
        $\mathscr{U}_R \leftarrow \mathscr{U}_R \parallel \mathtt{heap} := \{\mathscr{U}\}\mathtt{heap}$;    /* ...heap postcondition... */
        if $\mathtt{return}^m$ is assigned in $\mathscr{U}$ then
            $\mathscr{U}_R \leftarrow \mathscr{U}_R \parallel \mathtt{return}^m := \{\mathscr{U}\}\mathtt{return}^m$;    /* ...and result postcondition */
        end
        if $init$ then
            $R' \leftarrow R \cup \{(\Gamma \cup !\Delta, \mathscr{U}_R)\}$;    /* add normal postcondition */
            $redo' \leftarrow false$; /* only need to redo if a recursive call is encountered */
        else
            /* Normal postcondition may need refining (see Algorithm 16) */
            ...
        end
        /* update $PMC$ */
        $S' \leftarrow (S \setminus \{(m, init, (C', \mathscr{U}'), R)\}) \cup \{(m, init, (C', \mathscr{U}'), R')\}$;
        $PMC \leftarrow (PMC \setminus \{(M, S, redo)\}) \cup \{(M, S', redo')\}$;
        continue
    end
    ...
end
...
```

Algorithm 8: The part of method *eval* dealing with returns

The method *eval* deals with method calls returning exceptionally in a very similar manner to normal behavior. The update contains an assignment to $\mathtt{throw}^m$ based on the program statement, but otherwise the steps taken are almost identical. Algorithm 9 shows the steps involved for methods for which no recursive calls have yet been encountered.

Example 20. *Continuing Example 19, the sequents* (7.9), (7.10) *and* (7.11) *result in modification of the partial method contract mapping, such that the value of PMC is shown in* (7.12)*):*

$$\begin{aligned} PMC = \{\ (\{m\},\ (m, \mathit{true}, (\{v \leq 1\},\ \mathtt{heap} := h \parallel \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v), R),\ \mathit{false})\ \} \qquad (7.12)\\ \text{where } R = \{\ (\{v \doteq 0\},\ \mathtt{heap} := h \parallel \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{return}^m := 0),\\ (\{v \doteq 1\},\ \mathtt{heap} := h \parallel \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{return}^m := 1),\\ (\Gamma \cup \{v < 0\},\ \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{heap} := h' \parallel \mathtt{throw}^m := e)\ \} \end{aligned}$$

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with throws                                                    */
    if there is a seq $\in E$, where seq has the form:
        $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi$ method-frame(source=$m$,...) { throw $se$; ... } $\omega]\phi, \Delta$
    then
        $E \leftarrow E \setminus \{seq\}$;                      /* remove the sequent from $E$ */
        $(M, S, redo) \leftarrow PMC(m)$;          /* $PMC(m)$ will return a defined value */
        $(m, init, (C', \mathcal{U}'), R) \leftarrow S(m)$;
        /* create exceptional postcondition update with preconditions...       */
        $\mathcal{U}_R \leftarrow \mathtt{heap}^m := \{\mathcal{U}\}\mathtt{heap}^m \parallel \mathtt{p}_1^m := \{\mathcal{U}\}\mathtt{p}_1^m \parallel \ldots \parallel \mathtt{p}_n^m := \{\mathcal{U}\}\mathtt{p}_n^m$;
        $\mathcal{U}_R \leftarrow \mathcal{U}_R \parallel \mathtt{heap} := \{\mathcal{U}\}\mathtt{heap} \parallel \mathtt{throw}^m := \{\mathcal{U}\}se$;  /* ...and postconditions */
        if $init$ then
            $R' \leftarrow R \cup \{(\Gamma \cup !\Delta, \mathcal{U}_R)\}$;             /* add exceptional postcondition */
            $redo' \leftarrow false$; /* only need to redo if a recursive call is encountered */
        else
            /* Exceptional postcondition may need refining (see Algorithm 17)   */
            ...
        end
        /* update $PMC$                                                        */
        $S' \leftarrow (S \setminus \{(m, init, (C', \mathcal{U}'), R)\}) \cup \{(m, init, (C', \mathcal{U}'), R')\}$;
        $PMC \leftarrow (PMC \setminus \{(M, S, redo)\}) \cup \{(M, S', redo')\}$;
        continue
    end
    ...
end
...
```

Algorithm 9: The part of method *eval* dealing with throws

```
input : Method id $m$, flag $init$, set of postconditions $R$ and a sequent of one of the forms:
        $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathit{Class}.\texttt{m}(\texttt{x}_1,\ldots,\texttt{x}_n)\texttt{@}\mathit{Class};\ \omega]\phi, \Delta$
        $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{r=}\mathit{Class}.\texttt{m}(\texttt{x}_1,\ldots,\texttt{x}_n)\texttt{@}\mathit{Class};\ \omega]\phi, \Delta$
        $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{x}_1.\texttt{m}(\texttt{x}_2,\ldots,\texttt{x}_n)\texttt{@}\mathit{Class};\ \omega]\phi, \Delta$
        $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \texttt{r=x}_1.\texttt{m}(\texttt{x}_2,\ldots,\texttt{x}_n)\texttt{@}\mathit{Class};\ \omega]\phi, \Delta$
output: A set of sequents resulting from application of the specification

$E \leftarrow \emptyset$;
foreach $(C_R, \mathcal{U}_R) \in R$ do
    /* the modified heap is always an output of the method call */
    $h \leftarrow \{\mathcal{U}_R\}\texttt{heap}$;
    if $init = \mathit{false}$ then
        /* output values are always $\gamma$-symbols if not $init$ */
        $a \leftarrow$ the abstract element $h$ is a $\gamma$-symbol for
        $h \leftarrow$ a fresh $\gamma$-symbol for abstract element $a$;
    end
    /* all program variables other than heap remain unchanged by the method call */
    $\mathcal{U}' \leftarrow \mathcal{U} \parallel \texttt{heap}^m := \{\mathcal{U}\}\texttt{heap} \parallel \texttt{heap} := h$;
    p $\leftarrow$ the empty program statement “;” ;
    if $\texttt{throw}^m$ is assigned in $\mathcal{U}_R$ then
        p $\leftarrow$ the statement “throw exc;” ;
        exc $\leftarrow$ a fresh program variable of type Throwable;
        /* what was thrown from the method is also an output */
        $t \leftarrow \{\mathcal{U}_R\}\texttt{throw}^m$;
        if $init = \mathit{false}$ then
            $a \leftarrow$ the abstract element $t$ is a $\gamma$-symbol for;
            $t \leftarrow$ a fresh $\gamma$-symbol for abstract element $a$;
        end
        /* only the modified heap and what was thrown leave the method call */
        $\mathcal{U}' \leftarrow \mathcal{U}' \parallel \texttt{exc} := t$;
    else if the return value is stored in r in the active statement of the input sequent then
        $ret \leftarrow \{\mathcal{U}_R\}\texttt{return}^m$;
        if $init = \mathit{false}$ then
            $a \leftarrow$ the abstract element $ret$ is a $\gamma$-symbol for;
            $ret \leftarrow$ a fresh $\gamma$-symbol for abstract element $a$;
        end
        /* only the modified heap and the return value leave the method call */
        $\mathcal{U}' \leftarrow \mathcal{U}' \parallel \texttt{r} := ret$
    end
    /* instantiate constraints with new update */
    $C' \leftarrow C_R[\hat{\mathcal{U}}/\mathcal{U}']$;
    /* create a sequent from $C', \mathcal{U}'$, p and add it to the set of results */
    $E \leftarrow E \cup \{C' \Longrightarrow \{\mathcal{U}'\}[\pi\ \texttt{p}\ \omega]\phi\}$;
end
return $E$
```

Algorithm 10: Method *applyPost*

7.2.5 Applying the Method Contracts

Application of the generated method contracts is somewhat more complex than applying loop invariants. This is due in large part, however, to the different types of call and return possibilities. Algorithm 10 shows how a set of postconditions can be applied to a method call.

If the flag *init* is still set for the partial method contract, then no recursive calls have been encountered for the initial method call. In this case we can apply the postconditions in a relatively straightforward manner. Syntactic replacement of $\hat{\mathscr{U}}$ will also not take place in this case, as the initial constraints never contain $\hat{\mathscr{U}}$ and this is only added when abstracting the program states, which is only done once a recursive call has been encountered.

Example 21.

$$v \leq 1 \Longrightarrow \{\mathtt{heap} := h \parallel \mathtt{x} := v \parallel \mathtt{x}_1 := v\}[\pi\ \mathtt{x}_0\ \mathtt{=\ Math.fib(x}_1\mathtt{)@Math;\ f\ =\ x}_0\mathtt{;}\ \omega]\phi \tag{7.13}$$

Analysis of the sequent (7.13) *caused automatic generation of the postconditions R from Example 20. Using Algorithm 10 to calculate what applying a method contract with the given postconditions to the method call in sequent* (7.13) *results in, gives us the following three new sequents:*

$$v \doteq 0 \Longrightarrow \{\mathtt{x} := v \parallel \mathtt{x}_1 := v \parallel \mathtt{heap}^m := h \parallel \mathtt{heap} := h \parallel \mathtt{x}_0 := 0\}[\pi\ \mathtt{f\ =\ x}_0\mathtt{;}\ \omega]\phi \tag{7.14}$$

$$v \doteq 1 \Longrightarrow \{\mathtt{x} := v \parallel \mathtt{x}_1 := v \parallel \mathtt{heap}^m := h \parallel \mathtt{heap} := h \parallel \mathtt{x}_0 := 1\}[\pi\ \mathtt{f\ =\ x}_0\mathtt{;}\ \omega]\phi \tag{7.15}$$

$$\Gamma, v < 0 \Longrightarrow \{\mathtt{x} := v \parallel \mathtt{x}_1 := v \parallel \mathtt{heap}^m := h \parallel \mathtt{heap} := h' \parallel \mathtt{exc} := e\} \\ [\pi\ \mathtt{throw\ exc;\ f\ =\ x}_0\mathtt{;}\ \omega]\phi \tag{7.16}$$

In essence this is the same result as we would have gained by simply expanding the method call. This is as it should be, as no recursion was present to cause abstraction to be required.

Example 21 demonstrates an advantage of our automatic generation of specifications. Encountering the method call must cause the automatic generation of specifications to be triggered, as we cannot know at that point if recursion will be reached in the given program state. However, after generation of the specification we know for a fact whether any recursive calls were reached from the initial call. If this is not the case, our approach allows for full precision to be retained.

7.2.6 Nested Method Calls

If *eval* encounters a method call, what is to be done depends on whether this method is currently being investigated already, or if this method is new. If the method has not already caused a call to *generateContract*, this is initiated now. The goal is to either be able to generate a sound contract for this method call and apply it in order to then move on with the current execution of *eval*, or through analysis of this new method come across a mutual recursive call to the method currently being investigated. Algorithm 11 show the steps involved if the encountered method does not yet have a partial method contract mapping.

We have actually already quietly performed these steps in the examples so far, as the constructor call `IllegalArgumentException()` is a method call which causes the automatic generation of specifications just like any other call. The constructor may itself be recursive, or it may contain a loop or call a recursive method which requires the generation of specifications. As in this case the constructor for `IllegalArgumentException` was not recursive, we simply generated and applied a specification maintaining full precision.

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with method calls */
    if there is a $seq \in E$, where $seq$ has one of the forms:
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \mathit{Class}.\mathtt{m}(\mathtt{x}_1, \ldots, \mathtt{x}_n)\mathtt{@}\mathit{Class};\ \ \omega]\phi, \Delta$
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \mathtt{r=}\mathit{Class}.\mathtt{m}(\mathtt{x}_1, \ldots, \mathtt{x}_n)\mathtt{@}\mathit{Class};\ \ \omega]\phi, \Delta$
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \mathtt{x}_1.\mathtt{m}(\mathtt{x}_2, \ldots, \mathtt{x}_n)\mathtt{@}\mathit{Class};\ \ \omega]\phi, \Delta$
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \ \mathtt{r=x}_1.\mathtt{m}(\mathtt{x}_2, \ldots, \mathtt{x}_n)\mathtt{@}\mathit{Class};\ \ \omega]\phi, \Delta$
    then
        $E \leftarrow E \setminus \{seq\}$;                 /* remove the sequent from $E$ */
        $sig \leftarrow$ the signature for method m;
        $m \leftarrow \mathtt{m}(sig)\mathtt{@}\mathit{Class}$;                    /* create the method id */
        if $PMC(m)$ is defined then
            /* Deal with recursive method call (see Algorithm 13) */
            ...
        else
            /* generate a specification for $m$ */
            $(init, R) \leftarrow generateContract(\Gamma \cup !\Delta, \mathcal{U}, m, \bar{\mathtt{x}})$
        end
        /* apply the specification to $seq$ and add resulting sequents to $E$ */
        foreach $seq' \in applyPost(m, init, R, seq)$ do
            $E \leftarrow E \cup gatherSequents(seq')$;
        end
        continue
    end
    ...
end
...
```

Algorithm 11: The part of method *eval* dealing with method calls

7.2.7 Recursion

Encountering a recursive call, whether direct or mutually recursive, causes refinement of the preconditions as new program states are found in which the method is called. Additionally, if this was the first recursive call encountered, the postconditions are abstracted and joined. This is to ensure that only one abstracted normal postcondition and one abstracted exceptional postcondition for the method exist, in order to guarantee termination of the fixed point search. We cannot simply keep adding new postconditions to the set, otherwise a fixed point will never be reached. These steps are detailed in Algorithm 13.

In order to abstract and join the postconditions, we require the help of two additional methods: (i) *initialConstraints*, shown in Algorithm 12, to find the constraints which all postconditions can assume, removing the new constraints added due to branching in the method call, etc. and (ii) *joinPost*, shown in Algorithm 14, to accomplish the actual abstraction and joining of postconditions.

```
input : Constraint set $C$, possibly containing the placeholder update $\hat{\mathscr{U}}$.
output: A constraint set equal to $C$ without the constraints containing $\hat{\mathscr{U}}$.
$C' \leftarrow \emptyset$;
foreach $c \in C$ do
    if c does not contain the placeholder update $\hat{\mathscr{U}}$ then
        $C' \leftarrow C' \cup \{c\}$
    end
end
return $C'$
```

Algorithm 12: Method *initialConstraints*

Example 22. *Consider the sequent* (7.17)*, which creates the partial method contract mapping in* (7.18) *(with R shown in* (7.19) *like in Example 21) before eval encounters the recursive call in* (7.20)*.*

$$\Longrightarrow \{\texttt{heap} := h \parallel \texttt{x} := v \parallel \texttt{x}_1 := v\}[\pi\ \texttt{x}_0\ \texttt{= Math.fib(x}_1\texttt{)@Math; f = x}_0\texttt{;}\ \omega]\phi \tag{7.17}$$

$$PMC = \{\ (\{m\},\ (m, \mathit{true}, (\emptyset,\ \texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v), R),\ \mathit{false})\ \} \tag{7.18}$$

$$\begin{aligned} R = \{\ & (\{v \doteq 0\},\ \texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{return}^m := 0),\\ & (\{v \doteq 1\},\ \texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{return}^m := 1),\\ & (\Gamma \cup \{v < 0\},\ \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{heap} := h' \parallel \texttt{throw}^m := e)\ \} \end{aligned} \tag{7.19}$$

$$\begin{aligned} v \geq 2 \Longrightarrow & \{\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v \parallel \texttt{x}_1 := v \parallel \texttt{tmp}_1 := v - 2 \parallel \texttt{tmp}_2 := v - 2\}\\ & [\texttt{method-frame(result->return}^m\texttt{,source=}m\texttt{):\{ x}_0\texttt{ = Math.fib(tmp}_2\texttt{)@Math; ... \}}]\psi \end{aligned} \tag{7.20}$$

As PMC contains a mapping for m, the first step is to create an update from (7.20) *which can be joined with the precondition update from* (7.18)*. This leads to the update:*

$$\mathscr{U}_1 = \texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v - 2$$

Calling refine($\emptyset$, $\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v$, $\{v \geq 2\}$, $\texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := v{-}2$) *gives us the following refined constraint/update pair as new precondition for the partial method contract:*

$$C_2 = \emptyset \qquad\qquad \mathscr{U}_2 = \texttt{heap} := h \parallel \texttt{heap}^m := h \parallel \texttt{p}_1^m := \gamma_{\top,1}$$

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with method calls                                                    */
    if there is a seq $\in E$, where seq has one of the forms:
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathit{Class}.\mathtt{m}(\mathtt{x}_1,\ldots,\mathtt{x}_n)\mathtt{@}\mathit{Class};\ \omega]\phi, \Delta$
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{r=}\mathit{Class}.\mathtt{m}(\mathtt{x}_1,\ldots,\mathtt{x}_n)\mathtt{@}\mathit{Class};\ \omega]\phi, \Delta$
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{x}_1.\mathtt{m}(\mathtt{x}_2,\ldots,\mathtt{x}_n)\mathtt{@}\mathit{Class};\ \omega]\phi, \Delta$
            $\Gamma \Longrightarrow \{\mathcal{U}\}[\pi\ \mathtt{r=x}_1.\mathtt{m}(\mathtt{x}_2,\ldots,\mathtt{x}_n)\mathtt{@}\mathit{Class};\ \omega]\phi, \Delta$
    then
        ...
        if $PMC(m)$ is defined then
            /* Deal with recursive method call                                   */
            $(M, S, redo) \leftarrow PMC(m)$;        /* get partial method contract for $m$ */
            $(m, init, (C', \mathcal{U}'), R) \leftarrow S(m)$;
            /* create $\mathcal{U}_1$ which is joinable with $\mathcal{U}'$                          */
            $\mathcal{U}_1 \leftarrow \mathtt{heap} := \{\mathcal{U}\}\mathtt{heap} \parallel \mathtt{heap}^m := \{\mathcal{U}\}\mathtt{heap}^m;$
            $\mathcal{U}_1 \leftarrow \mathcal{U}_1 \parallel \mathtt{p}_1^m := \{\mathcal{U}\}\mathtt{x}_1 \parallel \ldots \parallel \mathtt{p}_n^m := \{\mathcal{U}\}\mathtt{x}_n;$
            /* refine $(C', \mathcal{U}')$                                              */
            $(C_2, \mathcal{U}_2) \leftarrow refine(C', \mathcal{U}', \Gamma \cup !\Delta, \mathcal{U}_1)$;
            if $(C_2, \mathcal{U}_2) \neq (C', \mathcal{U}')$ then
                /* contract needs refinement                                     */
                if $init$ then
                    $C \leftarrow initialConstraints(C')$;
                    $R' \leftarrow joinPost(m, C, R)$;   /* abstract and join postconditions */
                else $R' \leftarrow R$;
                /* update PMC with new specification, reset init and set redo   */
                $S' \leftarrow (S \setminus \{(m, init, (C', \mathcal{U}'), R)\}) \cup \{(m, false, (C_2, \mathcal{U}_2), R')\}$;
                $PMC \leftarrow (PMC \setminus \{(M, S, redo)\}) \cup \{(M, S', true)\}$;
            end
            /* merge mutually recursive methods on $PMC$ (see Algorithm 18)      */
            ...
            /* re-get (possibly refined) contract for $m$                        */
            $(\_, S, \_) \leftarrow PMC(m)$;
            $(m, init, \_, R) \leftarrow S(m)$
        else
            ...
        end
        /* apply the specification to $seq$ and add resulting sequents to $E$     */
        foreach $seq' \in applyPost(m, init, R, seq)$ do
            $E \leftarrow E \cup \{gatherSequents(seq')\}$;
        end
        continue
    end
    ...
end
...
```

Algorithm 13: The part of method *eval* dealing with recursive method calls

The method *joinPost* abstracts and joins precise postconditions, resulting in a set containing at most one normal postcondition and one exceptional postcondition. This is accomplished with the help of the methods *abstractPost* shown in Algorithm 15 and *refine*. The method *abstractPost* turns a precise postcondition into an abstract postcondition by keeping the input parameters precise, while finding smallest overapproximations for the output parameters (the program heap and possible returned or thrown value). Then a constraint set is added based on the initial constraints extended with instantiated invariant patterns containing the placeholder update $\hat{\mathcal{U}}$. This constraint set can be assumed to hold in the abstract update, as it held for the concrete realization it is based on.

Example 23. *Continuing Example 22, in order to abstract and join the postconditions we calculate the initial constraints $C = \emptyset$. The call to $joinPost(m, \emptyset, R)$ results in $R' = \{(C^R_1, \mathcal{U}^R_1), (C^R_2, \mathcal{U}^R_2)\}$, where:*

$$\begin{aligned}
C^R_1 &= \{\ \{\hat{\mathcal{U}}\}(\mathtt{heap} \doteq \mathtt{heap}^m),\ \{\hat{\mathcal{U}}\}(\mathtt{return}^m \doteq \mathtt{p}^m_1)\ \}\\
\mathcal{U}^R_1 &= \ \mathtt{heap}^m := h \parallel \mathtt{p}^m_1 := v \parallel \mathtt{heap} := h_1 \parallel \mathtt{return}^m := \gamma_{\geq,1}\\
C^R_2 &= \{\ \{\hat{\mathcal{U}}\}\Gamma_2\ \}\\
\mathcal{U}^R_2 &= \ \mathtt{heap}^m := h \parallel \mathtt{p}^m_1 := v \parallel \mathtt{heap} := h_2 \parallel \mathtt{throw}^m := e_2)
\end{aligned}$$

Here h_1 and h_2 are γ-symbols for heaps, expressing the heap h modified at no (non-created) locations or only at the locations needed for the newly created exception e respectively. Furthermore e_2 is a γ-symbol for objects, expressing the exact type `IllegalArgumentException`*, and Γ_2 contains constraints regarding* $\mathtt{heap}^m$, `heap` *and* $\mathtt{throw}^m$*.*

The partial method contract mapping is updated to become:

$$PMC = \{\ (\{m\},\ (m, \mathit{false}, (\emptyset,\ \mathtt{heap} := h \parallel \mathtt{heap}^m := h \parallel \mathtt{p}^m_1 := \gamma_{\top,1}), R'),\ \mathit{true})\ \}$$

Let us focus on the abstract postcondition for normal behavior: While both the return value and the heap have been abstracted, constraints have been added to remedy this coarse overapproximation. The abstract postcondition is based on the sequents leading to the method call returning normally so far and in each of these two cases (v is 0 or 1) both the output heap is equal to the input heap and the output value is equal to the parameter value. While this second constraint will not last, due to the fact that in general $\mathtt{fib}(x) \neq x$, the constraint regarding the heap is indeed valid in general for the normal behavior of `Math.fib`*.*

After the partial method contract mapping has been updated due to a recursive call, the call itself is replaced by applying the partial method contract gathered so far. This is needed in order to reach program points occurring after the recursive call, which can be further recursive calls, method returns or exception throwing, all of which must be taken into account.

As the method call has been determined to be recursive, we must use fresh γ-symbols for the output values, rather than the actual γ-symbols contained in the postconditions. The reason for this is simple: Let f be some method under consideration. Then we are not interested in the value of $f(\gamma_{a_i,j})$, but rather the value of $f(\gamma_{a_i,z})$ for all $z \in \mathbb{Z}$. This means in particular, that the return value for $f(\gamma_{a_i,j})$ being $\gamma_{a_k,l}$ does not imply that $\gamma_{a_k,l}$ is the return value for *all* $f(\gamma_{a_i,z})$, but only that for each $z \in \mathbb{Z}$ there is a $z' \in \mathbb{Z}$, such that $f(\gamma_{a_i,z}) = \gamma_{a_k,z'}$.

```
input  : Method id $m$, with initial constraint set $C$ and a set of constraint/update pairs $R$
         expressing return and/or throw postconditions.
output : A set of abstracted constraint/update pairs, maximum of one for returns and one
         for throws.

$Returns \leftarrow \emptyset$;
$Throws \leftarrow \emptyset$;
foreach $(C_R, \mathcal{U}_R) \in R$ do
    if $\texttt{throw}^m$ is assigned in $\mathcal{U}_R$ then
        if $Throws \neq \emptyset$ then
            $(C'_R, \mathcal{U}'_R) \leftarrow$ the element in $Throws$;
            $Throws \leftarrow \{refine(C'_R, \mathcal{U}'_R, C_R, \mathcal{U}_R)\}$
        else $Throws \leftarrow \{abstractPost(C, C_R, \mathcal{U}_R)\}$;
    else if $Returns \neq \emptyset$ then
        $(C'_R, \mathcal{U}'_R) \leftarrow$ the element in $Returns$;
        $Returns \leftarrow \{refine(C'_R, \mathcal{U}'_R, C_R, \mathcal{U}_R)\}$
    else $Returns \leftarrow \{abstractPost(C, C_R, \mathcal{U}_R)\}$;
end
return $Returns \cup Throws$
```

Algorithm 14: Method *joinPost*, to abstract and join postconditions.

```
input  : Method id $m$ with initial constraint set $C$ and constraint/update pair $(C_2, \mathcal{U}_2)$,
         where $\hat{\mathcal{U}}$ does not appear in either $C$ or $C_2$.
output : An abstracted constraint/update pair.

/* gather the program variables which are outputs of the method call      */
$O \leftarrow \{\texttt{heap}\}$;
if $\texttt{throw}^m$ is assigned in $\mathcal{U}_2$ then
    $O \leftarrow O \cup \{\texttt{throw}^m\}$
else if $\texttt{return}^m$ is assigned in $\mathcal{U}_2$ then
    $O \leftarrow O \cup \{\texttt{return}^m\}$
end
/* keep the inputs                                                         */
$\mathcal{U}' \leftarrow \texttt{p}_1^m := \{\mathcal{U}_2\}\texttt{p}_1^m \parallel \ldots \parallel \texttt{p}_n^m := \{\mathcal{U}_2\}\texttt{p}_n^m \parallel \texttt{heap}^m := \{\mathcal{U}_2\}\texttt{heap}^m$;
/* abstract the outputs                                                    */
foreach $\texttt{x} \in O$ do
    $\mathcal{A} \leftarrow$ the abstract domain for the type of $\texttt{x}$;
    $a \leftarrow$ the smallest abstract element of $\mathcal{A}$, where $C_2 \Longrightarrow \chi_a(\{\mathcal{U}_2\}\texttt{x})$ is valid;
    $\gamma_{a,j} \leftarrow$ a fresh $\gamma$-symbol for the abstract element $a$;
    $\mathcal{U}' \leftarrow \mathcal{U}' \parallel \texttt{x} := \gamma_{a,j}$
end
foreach invariant pattern $p$ do
    $c' \leftarrow$ strongest invariant for $p$, such that $C_2 \Longrightarrow c'[\hat{\mathcal{U}}/\mathcal{U}_2]$ is valid;
    $C' \leftarrow C' \cup \{c'\}$
end
return $(C', \mathcal{U}')$
```

Algorithm 15: Method *abstractPost* to create an abstract constraint/update pair.

Example 24. *Continuing Example 23, we apply the postconditions in R' to the sequent:*

$$v \geq 2 \Longrightarrow \{\mathtt{heap} := h \parallel \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{x}_1 := v \parallel \mathtt{tmp}_1 := v - 2 \parallel \mathtt{tmp}_2 := v - 2\}$$
$$[\mathtt{method\text{-}frame(result\text{-}\!>\!return}^m\mathtt{,source=}m\mathtt{):\{\ x_0\ =\ Math.fib(tmp_2)@Math;\ \ldots\ \}}]\psi$$

After symbolic execution of the resulting sequents in gatherSequents, the following (somewhat simplified) sequents are added to E, where Γ' contains constraints regarding h, h'_2 and e'_2, while h'_1, h'_2, e'_2 are fresh γ-symbols for the same abstract elements as h_1, h_2, e_2:

$$v \geq 2, h'_1 \doteq h, \gamma_{\geq,2} \doteq v \Longrightarrow \{\mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{x}_1 := v \parallel \mathtt{heap} := h'_1 \parallel \mathtt{x}_0 := \gamma_{\geq,2} \parallel \mathtt{tmp} := v - 1\}$$
$$[\mathtt{method\text{-}frame(result\text{-}\!>\!return}^m\mathtt{,source=}m\mathtt{):\{\ y_0\ =\ Math.fib(tmp)@Math;\ \ldots\ \}}]\psi \tag{7.21}$$

$$\Gamma', v \geq 2 \Longrightarrow \{\mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{x}_1 := v \parallel \mathtt{heap} := h'_2 \parallel \mathtt{exc} := e'_2\}$$
$$[\mathtt{method\text{-}frame(result\text{-}\!>\!return}^m\mathtt{,source=}m\mathtt{):\{\ throw\ exc;\ \ldots\ \}}]\psi \tag{7.22}$$

The sequent (7.21) contains a recursive call, but refinement reveals that the partial method contract is already applicable for this call and so application of the postcondition and symbolic execution in gatherSequents leads to:

$$v \geq 2,\ h'_1 \doteq h,\ h''_1 \doteq h'_1,\ \gamma_{\geq,2} \doteq v,\ \gamma_{\geq,3} \doteq v \ \Longrightarrow$$
$$\{\mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{x}_1 := v \parallel \mathtt{x}_0 := \gamma_{\geq,2} \parallel \mathtt{heap} := h''_1 \parallel \mathtt{y}_0 := \gamma_{\geq,3} \parallel \mathtt{return}^m := \gamma_{\geq,2} + \gamma_{\geq,3}\}$$
$$[\mathtt{method\text{-}frame(source=}m\mathtt{):\{\ \}}]\psi \tag{7.23}$$

Once a method call has been deemed recursive, we can no longer simply add new postconditions when normal or exceptional termination of the method is reached. Instead, the new postcondition is used to refine the existing normal or exceptional postcondition. If a matching postcondition type does not already exist, then the new postcondition is instead abstracted and this abstract postcondition is added. Thus we continue to ensure that only at most one normal postcondition and at most one exceptional postcondition exist for the partial method contract for a recursive method. Details on these steps for normal postconditions are in Algorithm 16 and for exceptional postconditions in Algorithm 17. The differences between dealing with normal and exceptional postconditions are again quite minimal.

Example 25. *The sequents (7.22) and (7.23) from Example 24 cause postcondition refinement. In the case of the exceptional postcondition, the result of refine does not change, so no further action is taken. This is in fact a fixed point for the exceptional behavior: an exception of type* `IllegalArgumentException` *is thrown, it is stored on the heap and its fields are initialized. Nothing more can be gained from this abstract exceptional postcondition.*

Refinement of the normal postcondition $(C'_R, \mathcal{U}'_R)$ with the postcondition $(C_R, \mathcal{U}_R)$ gained from sequent (7.23), however, leads to the new postcondition $(C_1, \mathcal{U}_1)$, where:

$$C'_R = \{\ \{\hat{\mathcal{U}}\}(\mathtt{heap} \doteq \mathtt{heap}^m),\ \{\hat{\mathcal{U}}\}(\mathtt{return}^m \doteq \mathtt{p}_1^m)\ \}$$
$$\mathcal{U}'_R = \ \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{heap} := h_1 \parallel \mathtt{return}^m := \gamma_{\geq,1}$$

$$C_R = \{\ v \geq 2,\ h'_1 \doteq h,\ h''_1 \doteq h'_1,\ \gamma_{\geq,2} \doteq v,\ \gamma_{\geq,3} \doteq v\ \}$$
$$\mathcal{U}_R = \ \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{heap} := h''_1 \parallel \mathtt{return}^m := \gamma_{\geq,2} + \gamma_{\geq,3}$$

$$C_1 = \{\ \{\hat{\mathcal{U}}\}(\mathtt{heap} \doteq \mathtt{heap}^m)\ \}$$
$$\mathcal{U}_1 = \ \mathtt{heap}^m := h \parallel \mathtt{p}_1^m := v \parallel \mathtt{heap} := h'_3 \parallel \mathtt{return}^m := \gamma_{\geq,4}$$

The method eval now has no more sequents to consider and so returns to generateContracts. As the redo-flag has been set for for the method identifier m in the partial method contract mapping, a further iteration of the fixed point search will be triggered. In this new call to eval, all recursive method calls can be replaced with their postconditions without modifying the precondition. Further, the postconditions will not need to be modified, either. A fixed point has therefore been found for the program state of the initial method call and an overapproximation of all program states in which recursive calls can be reached.

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with returns */
    if there is a seq $\in E$, where seq has the form:
        $\Gamma \Longrightarrow \{\mathscr{U}\}[\pi$ method-frame(source=$m$,...) { } $\omega]\phi, \Delta$
    then
        ...
        /* $\mathscr{U}_R$ is the normal postcondition update calculated from seq */
        ...
    else
        /* init is false, normal postcondition may need refining */
        if $R$ contains a $(C'_R, \mathscr{U}'_R)$, where throw$^m$ is not in $\mathscr{U}'_R$ then
            $(C_1, \mathscr{U}_1) \leftarrow refine(C'_R, \mathscr{U}'_R, \Gamma \cup !\Delta, \mathscr{U}_R)$;
            if $(C_1, \mathscr{U}_1) = (C'_R, \mathscr{U}'_R)$ then
                continue            /* no refinement to specification needed */
            end
            /* need to update normal postcondition */
            $R' \leftarrow (R \setminus \{(C'_R, \mathscr{U}'_R)\}) \cup \{(C_1, \mathscr{U}_1)\}$
        else
            /* get constraints of initial method call */
            $C \leftarrow initialConstraints(C')$;
            /* add abstracted normal postcondition */
            $R' \leftarrow R \cup \{abstractPost(C, \Gamma \cup !\Delta, \mathscr{U}_R)\}$
        end
        $redo' \leftarrow true$;   /* specification was changed, fixed point not found */
    end
    /* update $PMC$ */
    $S' \leftarrow (S \setminus \{(m, init, (C', \mathscr{U}'), R)\}) \cup \{(m, init, (C', \mathscr{U}'), R')\}$;
    $PMC \leftarrow (PMC \setminus \{(M, S, redo)\}) \cup \{(M, S', redo')\}$;
    continue
    end
    ...
end
...
```

Algorithm 16: The part of method *eval* dealing with returns for a recursive method

```
...
while $E \neq \emptyset$ do
    ...
    /* Deal with throws */
    if there is a $seq \in E$, where $seq$ has the form:
        $\Gamma \Longrightarrow \{\mathscr{U}\}[\pi$ method-frame(source=$m$,...) { throw $se$; ... } $\omega]\phi, \Delta$
    then
        ...
        /* $\mathscr{U}_R$ is the exceptional postcondition update calculated from $seq$ */
        ...
        else
            /* $init$ is false, exceptional postcondition may need refining */
            if $R$ contains a $(C'_R, \mathscr{U}'_R)$, where throw$^m$ is in $\mathscr{U}'_R$ then
                $(C_1, \mathscr{U}_1) \leftarrow refine(C'_R, \mathscr{U}'_R, \Gamma \cup !\Delta, \mathscr{U}_R)$;
                if $(C_1, \mathscr{U}_1) = (C'_R, \mathscr{U}'_R)$ then
                    continue        /* no refinement to specification needed */
                end
                /* need to update exceptional postcondition */
                $R' \leftarrow (R \setminus \{(C'_R, \mathscr{U}'_R)\}) \cup \{(C_1, \mathscr{U}_1)\}$
            else
                /* get constraints of initial method call */
                $C \leftarrow initialConstraints(C')$;
                /* add abstracted exceptional postcondition */
                $R' \leftarrow R \cup \{abstractPost(C, \Gamma \cup !\Delta, \mathscr{U}_R)\}$
            end
            $redo' \leftarrow true$;   /* specification was changed, fixed point not found */
        end
        /* update $PMC$ */
        $S' \leftarrow (S \setminus \{(m, init, (C', \mathscr{U}'), R)\}) \cup \{(m, init, (C', \mathscr{U}'), R')\}$;
        $PMC \leftarrow (PMC \setminus \{(M, S, redo)\}) \cup \{(M, S', redo')\}$;
        continue
    end
    ...
end
...
```

Algorithm 17: The part of method *eval* dealing with throws for a recursive method

7.2.8 Mutual Recursion

As was mentioned in Section 7.2.3, the method *generateContract* is designed in order to also be able to generate method contracts for mutually recursive methods. Let us briefly sketch the idea: If a method f calls a method g, which calls a method h, then during analysis of the method body for h the partial method contract mapping is:

$$\begin{aligned} PMC = \{ \ & (\{f\},\ \{(f, init_f, pre_f, POST_f)\},\ redo_f), \\ & (\{g\},\ \{(g, init_g, pre_g, POST_g)\},\ redo_g), \\ & (\{h\},\ \{(h, init_h, pre_h, POST_h)\},\ redo_h) \ \} \end{aligned}$$

If a partial method contract can be generated for the method h, then it is returned and $PMC(h)$ is removed from PMC. This removal allows multiple calls to the same non-recursive method to occur without triggering abstraction, to ensure high precision specifications. Even for recursive methods this can enhance the precision, as we might generate two vastly different method contracts if a method is called with a positive or negative input. If these were joined, we lose quite a bit of precision in the precondition, which could cause loss of precision in the postconditions as well.

But what if h calls f or g? To begin with, we must treat this recursive call to f or g as usual. Additionally, however, we need to establish in these cases that multiple methods have the capability of calling each other and therefore changes to the partial method contract of one method may also influence the partial method contracts of other methods. There are two questions to answer:

1. Which methods exactly are involved in the mutual recursion?

2. How can we ensure that the fixed points gathered for all these methods interact appropriately?

The answer to the first question is on the method frame stack. As we can trace back through the stack, we can gather all methods involved in the mutual recursion. If the method stack has h as its top element and g right below it, then a call to g within h only involves these two methods. If, however, f is called from within h, then we must gather all methods on the stack up to and including f. For if f calls g calls h calls f, then all three methods are mutually recursive.

The solution for the second question is implemented in the following manner:

1. The partial method contract mapping maps *sets* of method identifiers to *sets* of partial method contracts. This allows mutually recursive methods to be grouped. As there is only one *redo*-flag per element in the partial method contract mapping, this ensures that any member of the group setting this flag will suffice.

2. The fixed point search in *generateContract* checks *all* of the partial method contracts in $PMC(m)$ for the method m under consideration. This means that not only m itself, but all mutually recursive methods grouped with m will be tested before it is determined if a fixed point is found.

3. The treatment for sequents containing a method call as active statement perform the steps in Algorithm 18 after refining the precondition for m and before applying the postcondition. This ensures that mutually recursive method calls are recognized as such.

```
...
/* merge mutually recursive methods on $PMC$                                  */
$M' \leftarrow \mathbf{get}(MFS, m)$;          /* get all methods involved in recursion */
if $M' \not\subseteq M$ then
    $joinPMC(M \cup M')$;                                 /* merge all on PMC */
end
...
```

Algorithm 18: The part of method *eval* dealing with mutual recursive method calls

```
global: A partial method contract mapping $PMC$.
input : Set of method ids $M$
$M' \leftarrow M$;
$S' \leftarrow \emptyset$;
while $M \neq \emptyset$ do
    $m' \leftarrow$ some element of $M'$;
    $(M_1, S_1, redo_1) \leftarrow PMC(m')$;
    $PMC \leftarrow PMC \setminus \{(M_1, S_1, redo_1)\}$;          /* remove $M_1$ from $PMC$ */
    $M \leftarrow M \setminus M_1$;                                  /* remove $M_1$ from $M$ */
    $S' \leftarrow S' \cup S_1$;      /* $S'$ now contains contracts for all methods in $M_1$ */
end
$PMC \leftarrow PMC \cup \{(M', S', true)\}$;                   /* add new entry to $PMC$ */
```

Algorithm 19: Method *joinPMC*, joining elements of *PMC* into a single element

Algorithm 19 performs the actual joining of members of the partial method contract mapping. All members containing partitions which have members belonging to the set of newly recognized mutually recursive methods are joined into one member.

Example 26. *Let the method frame stack and partial method contract mapping be:*

$$MFS = \{\ \{f, g, h\},\ \{g, h\}\ , \{h\}\ \}$$

$$\begin{aligned} PMC = \{\ \ & (\{f, g\},\ \{(f, init_f, pre_f, POST_f),\ (g, init_g, pre_g, POST_g)\},\ redo_{fg}),\\ & (\{h\},\ \{(h, init_h, pre_h, POST_h)\},\ redo_h)\ \ \} \end{aligned}$$

If a method call to g is encountered, then first the element

$$(\{f, g\},\ \{(f, init_f, pre_f, POST_f),\ (g, init_g, pre_g, POST_g)\},\ redo_{fg})$$

is retrieved and init and pre_g updated to false and pre'_g. Then the method frame stack is checked, to reveal that $\{g, h\}$ is the minimal set of method identifiers that must be grouped. As $\{g, h\} \not\subseteq \{f, g\}$, we call $joinPMC(\{f, g, h\})$, which results in the partial method contract mapping being modified to:

$$PMC = \{(\{f, g, h\}, \{(f, init_f, pre_f, POST_f),\ (g, false, pre'_g, POST_g),\ (h, init_h, pre_h, POST_h)\}, true)\}$$

7.2.9 Recursive Calls Within Loops

In the case that a recursive call is encountered within a loop, i.e. during analysis of a loop in *eval* called from *generateLoopInvariant*, which itself was called during analyisis of a method call in *eval* called from *generateContract*, we first perform all necessary steps on the recursive call:

1. possible refinement of the partial method contract's precondition,
2. possible grouping of partial method contracts due to mutual recursion,
3. application of the partial method contract's postconditions on the sequent containing the recursive call, and finally
4. gathering the sequents of open branches in the symbolic execution tree resulting from symbolic execution of the sequents resulting from these applications.

We then continue analysis of the loop. This can lead to the same recursive calls being encountered in multiple iterations of the fixed point search for the loop. Often this will be in a different program state. Once a fixed point for the loop invariant has been found, it will be applied to the loop upon returning from *generateLoopInvariant*. This application will then lead in *eval* to new sequents returning from the method containing the loop (normally or exceptionally). These may further modify the partial method contract for the method call. If no modification took place, either of the precondition due to the recursive call or the postconditions, then no further steps are required and the postconditions will be returned and applied. In the general case, however, where the partial method contract has been modified and therefore requires a new iteration of the fixed point search, the loop for which we have already generated a loop invariant will be encountered again, this time potentially with a different program states. Therefore, similar to the way nested inner loops are dealt with in Section 7.1.2, this loop must also have its loop invariant re-generated. Even in the case where the program state is the same, the loop needs to be re-analyzed, as the postconditions for the method may have changed, resulting in modified behavior of the loop body.

Note on Method Overriding

This approach works not only for static methods and private instance methods, which do not have the capability to be overridden, but also for non-private instance methods which can be overridden. This is due to the fact that we use method identifiers to determine which actual method implementation is being called before considering if this is a recursive call. Given the Java `class` definitions in Listing 7.2, we note the following:

1. The method identifiers `f(C×int→int)@C` and `f(E×int→int)@E` are different.
2. A call of `f(E×int→int)@E` will not reach any recursive calls and therefore a fully precise postcondition will be generated.
3. A call of `f(C×int→int)@C` may or may not reach a recursive call. In particular, even in the case that `x` $>$ 0 and `o` $\neq$ `null`, a recursive call might not be reached: if `o` is of type `E` (or a subtype thereof not overriding method `f`), then the call to `o.f(x-1)` is not a recursive call, but rather a call of the *different, non-recursive* method identifier `f(E×int→int)@E`. Therefore in this case the call of `f(C×int→int)@C` also need not be abstracted and a fully precise postcondition will be generated and applied.

```
class C {
  int f(int x) {
    if (x <= 0)
      return 0;
    return 1 + o.f(x-1);
  }

  C o;
}
```

```
class E extends C {
  int f(int x) {
    if (x <= 0)
      return 0;
    return x;
  }
}
```

Listing 7.2: Classes demonstrating method overriding

Our approach cannot fully support an open world assumption on additional types when generating specifications for non-private instance methods, however, as it relies on the use of the calculus rule methodCall in order to let a call `o.f(x-1)` be split into the various calls to different method identifiers. Use of the methodCall rule assumes a closed world in this regard, however.

Note on Method Frames

In our analysis we do not examine each recursive call in its actual call environment, but rather create a new sequent with the method call expanded at top-level. This is required, as the set of all possible call environments for recursive calls could be infinite. As such, we cannot allow the method call to access all method-frames, as otherwise our sandbox would perform differently than the actual environment. This is not much of a problem, as Java does not allow direct access to the method-frames. However, there are some Java library methods which do allow indirect access to method-frames, such as methods to print the stack trace. Momentarily, this is not part of Java dynamic logic. However, if it were to be added, we would require a safe overapproximation of the method-frames in which a call is made.

8 Related Work

In this chapter we discuss related work in the fields of automatic invariant generation, abstract interpretation, array and heap abstraction, as well as automatic generation of method summaries. We contrast these with the solutions chosen in this dissertation.

Abstract Interpretation: There are various approaches to automatic generation of specifications using abstract interpretation (see [23] for a survey). As noted in [28], although abstract interpretation relies on abstract domains being lattices, restriction in this manner is quite costly on precision. For this reason many approaches use non-lattice abstract domains. Examples of these are *wrapped intervals* [49] and the array segmentations proposed in [31, 35, 17]. Non-lattice abstract domains, however, are not directly usable with abstract interpretation. Ad-hoc introduction of *quasi-joins* can allow the integration, but causes other problems [28]. *Predicate abstraction* [7, 8, 38, 63] is a quite efficient technique, using predicates about the values of program variables to replace these.

We rely on abstract domains which are lattices. Our abstract domain for integers $\mathscr{A}^{\texttt{int}}$, while not quite as expressive as wrapped intervals, has the advantage of the result of each join being clearly defined. By using abstract domains, but adding refinement invariants, we keep precision high while not needed to deal with the problems of non-lattice domains. We do not perform abstraction of program variables, but rather their states.

Array Abstraction and Invariants: There has been much research in the field of array abstraction. Abstract interpretation results range from simple *array smashing* [11], where all elements of the array are treated as one abstract element, to the more detailed array segmentations of [17], where arrays can be partitioned into numerous different ranges. Based on *linear loop-dependent scalars* [20] or the various properties (increasing, dense, etc.) for scalars in [46], or utilizing *range predicates* [43], allows invariants for ranges within arrays to be automatically generated. Many of these approaches only allow for contiguous ranges. Other approaches to array abstraction are to use templates to introduce quantified formulas from quantifier-free elements [33], or by abstracting the program itself by replacing the array with multiple array slices as in [35].

With the approach outlined in this dissertation we can handle both contiguous as well as affine ranges when generating instantiations of *invariant patterns* for array updates. We do not require the abstraction of the program itself, but rather only the abstraction of the program states, thereby potentially keeping higher precision. Our use of invariant patterns, while different from the concept of templates in [33], results similarly in the generation of invariants of a predefined form.

Heap Abstraction: Some interesting abstraction techniques for heaps are the various *points-to-analyses* [39] and the related *shape analysis* [51, 44], as well as *canonical abstraction* [52]. With these approaches it is possible to automatically perform abstraction on (parts of) the program heap, forming abstract members which express certain shapes, reachability issues, etc.

Our heap abstraction is limited to anonymizing a smallest overapproximation of the location sets modified and refining this with instantiations of invariant patterns. While this approach is not as precise as shape analysis for reasoning about lists and trees, it is firmly anchored in an abstract domain expressible as a lattice, with a sound non-trivial widening operator.

Automatic Generation of Loop Invariants and Method Summaries: In general, the approaches to automatic generation of loop invariants and method summaries are either restricted to an academic toy language, require modification of the program to, for example, remove recursion [13] or replace concrete types with abstract ones [7, 8, 38], use bottom-up optimizations unfitting for mutual recursive methods [26, 64], or analyze only *some* of the dynamic traces of a program [25]. Some interesting approaches [50, 42, 47, 18, 27, 3] (can) use provided annotations to guide generation of loop invariants and pre- and postconditions for non-recursive methods. The approaches in [50, 3] also utilize symbolic execution.

Our automatic generation of specifications is based on a real-world language and requires no modification of the program. The top-down approach allows easy integration of mutual recursion and symbolic execution allows consideration of *all* program traces. As some of the use cases for our approach do not provide annotations, we cannot use these to guide our generation of specifications. Furthermore, all known approaches which use provided annotations in the automatic generation of specifications cannot deal with recursive methods.

9 Conclusion

This dissertation has demonstrated how abstract interpretation and symbolic execution can be combined in order to automatically generate loop invariants and method contracts in real-life programming languages containing, among others, nested loops, mutual recursion, exceptional behavior and non-standard control flows.

Useful abstract domains for integers, booleans, objects and heaps have been introduced. In particular, the abstract domain for heaps is to our knowledge the first abstract domain for heaps relying on an infinite-height lattice with non-trivial widening operator which tries to retain full precision of which locations on the heap have been modified if these locations are not responsible for the widening.

In order to retain higher precision than with only an abstraction using abstract domains, it has been shown how automatic instantiations of *invariant patterns* can counteract the coarse overapproximations of an abstract domain, while still retaining a terminating fixed point algorithm. These invariant patterns allow for relational invariants, strengthening of the heap abstraction and invariants for array contents, by classification for contiguous or affine ranges within the array and the complementing range.

Non-standard control flows within loops complicate analysis and application of loop invariant rules. This dissertation uncovered problems with two existing attempts at loop invariant rules for non-standard control flows in Java dynamic logic: (i) by the introduction of a multitude of new modalities, and (ii) by program transformation of the loop body. The introduction of *loop scopes* into Java dynamic logic allows gathering the necessary sequents for automatic generation of loop invariants. Using an extension of loop scopes, a sound loop invariant rule for Java dynamic logic has been proposed, which does not have the problems the other attempts had and furthermore relies neither on the introduction of new modalities, nor on the transformation of the loop body, both of which are error-prone. Furthermore, this extension of loop scopes can also be used to replace the loop unwinding rule.

Finally, the entire process of automatic generation and application of sound loop invariants and method contracts has been described. This process is capable of dealing with the possible presence of nested loops, mutual recursion and recursive calls from within a loop.

A case study [22] has illustrated how an implementation of this approach can help in the automatic analysis of programs for information flow properties. Further use case scenarios are in helping with the verification of partially specified code, for example by integration with [37], and integration with an IDE in order to present the programmer with specifications generated on-the-fly.

The automatic generation of specifications outlined in this dissertation uses the box modality $[\cdot]$ which covers partial correctness. This could be extended to total correctness, using the diamond modality $\langle\cdot\rangle$. A simple solution would be to use the calculus rule splitTermination in (10.1) in order to separate the total correctness requirement into (i) proving partial correctness and (ii) proving termination. For the partial correctness, specifications can be automatically generated as before. For the termination proof, a program can be generated from the sequent and passed to an existing tool for termination analysis, such as AProVE [30] or COSTA [4]. A more comprehensive solution would involve proving both aspects of total correctness at the same time, as termination proofs often calculate helpful invariants as side effects which could be used in the proof of partial correctness, while for some programs knowledge of partial correctness may be required in order to prove termination.

$$\mathsf{splitTermination}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathrm{p}]\phi, \Delta \qquad \Gamma \Longrightarrow \{\mathcal{U}\}\langle\mathrm{p}\rangle\mathrm{true}, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}\langle\mathrm{p}\rangle\phi, \Delta} \tag{10.1}$$

Invariant patterns could be introduced for reachability concerns on the heap, using shape analysis techniques to allow better invariants for structures such as lists and trees. The invariant patterns for arrays could be further enhanced to allow multi-range partitions in order to generate strong specifications for methods such as quicksort, which modify arrays at multiple (contiguous) ranges.

While not a requirement in all use cases for automatic generation of specifications, often annotations for at least parts of the source code are provided. Using ideas from [50, 3] these annotations could be utilized to help guide the generation of stronger loop invariants. Furthermore, the use of annotations to guide generation of method contracts for (mutual) recursive methods could be investigated.

Integration of the information flow tracking ideas in [61] into the framework for automatic generation of specifications could lead to the automatic generation of information flow properties, or at the least to the automatic generation of invariants regarding explicit and implicit dependencies for program variables and array elements.

Bibliography

[1] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The key platform for verification and analysis of java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments: 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, pages 55–71. Springer International Publishing, Cham, 2014.

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification – The KeY Book: From Theory to Practice*. Springer International Publishing, Cham, 2016.

[3] Wolfgang Ahrendt, Laura Kovács, and Simon Robillard. Reasoning about loops using vampire in key. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 9450 of *Lecture Notes in Computer Science*, pages 434–443. Springer Berlin Heidelberg, 2015.

[4] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Resource usage analysis and its application to resource certification. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 258–288. Springer Berlin Heidelberg, 2009.

[5] Stephan Arlt, Philipp Rümmer, and Martin Schäf. Joogie: From java through jimple to boogie. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 3–8, New York, NY, USA, 2013. ACM.

[6] Michael Francis Atiyah and I. G. MacDonald. *Introduction to commutative algebra.* Addison-Wesley-Longman, 1969.

[7] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer, 2000.

[8] Thomas Ball and SriramK. Rajamani. The slam toolkit. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.

[9] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* Number 4334 in LNCS. Springer, 2007.

[10] Garrett Birkhoff. Lattice theory. In *Colloquium Publications*, volume 25. Amer. Math. Soc., 3. edition, 1967.

[11] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough, editors, *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002.

[12] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In *7th Intl. Symposium on Formal Methods for Components and Objects (FMCO 2008)*, volume 5751 of *LNCS*, pages 247–277. Springer, 2009.

[13] Yu-Fang Chen, Chiao Hsieh, Ming-Hsien Tsai, Bow-Yaw Wang, and Farn Wang. Verifying recursive programs using intraprocedural analyzers. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis*, volume 8723 of *LNCS*, pages 118–133. Springer, 2014.

[14] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[15] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[16] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, PLILP '92, pages 269–295, London, UK, UK, 1992. Springer-Verlag.

[17] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of the 38th Symposium on Principles of Programming Languages*, POPL '11, pages 105–118. ACM, 2011.

[18] Guido de Caso, Diego Garbervetsky, and Daniel Gorín. Reducing the number of annotations in a verification-oriented imperative language. *CORR*, abs/1011.3407, 2010.

[19] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970.

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 246–266. Springer, 2010.

[21] Quoc Huy Do, Richard Bubel, and Reiner Hähnle. Exploit generation for information flow leaks in object-oriented programs. In Hannes Federrath and Dieter Gollmann, editors, *ICT Systems Security and Privacy Protection*, volume 455 of *IFIP Advances in Information and Communication Technology*, pages 401–415. Springer International Publishing, 2015.

[22] Quoc Huy Do, Eduard Kamburjan, and Nathan Wasser. Towards fully automatic logic-based information flow analysis: An electronic-voting case study. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust: 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, pages 97–115, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[23] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1165–1178, July 2008.

[24] Vijay D'Silva and Caterina Urban. Abstract interpretation as automated deduction. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 450–464. Springer International Publishing, 2015.

[25] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.

[26] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517. Springer, 2001.

[27] Carlo A. Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation*, volume 6300 of *Lecture Notes in Computer Science*, pages 277–300. Springer, 2010.

[28] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Abstract interpretation over non-lattice abstract domains. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, pages 6–24, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[29] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* North Holland, 1969.

[30] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. 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.

[31] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. *SIGPLAN Not.*, 40(1):338–350, January 2005.

[32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison Wesley, 2nd edition, 2000.

[33] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In George C. Necula and Philip Wadler, editors, *POPL*, pages 235–246. ACM, 2008.

[34] Reiner Hähnle, Nathan Wasser, and Richard Bubel. Array abstraction with symbolic pivots. In Erika Ábrahám, Marcello Bonsangue, and Einar Broch Johnsen, editors, *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 104–121. Springer International Publishing, Cham, 2016.

[35] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *PLDI*, pages 339–348. ACM, 2008.

[36] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic.* MIT Press, Cambridge, MA, USA, 2000.

[37] Martin Hentschel, Stefan Käsdorf, Reiner Hähnle, and Richard Bubel. An interactive verification tool meets an IDE. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, pages 55–70, 2014.

[38] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70. ACM, 2002.

[39] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[40] Marieke Huisman and Bart Jacobs. Java program verification via a hoare logic with abrupt termination. In *Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering: Held As Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000*, FASE '00, pages 284–303, London, UK, UK, 2000. Springer-Verlag.

[41] Bart Jacobs. A formalisation of java's exception mechanism. In David Sands, editor, *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings*, pages 284–301, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[42] M. Janota. Assertion-based loop invariant generation. In *Proceedings of the 1st International Workshop on Invariant Generation, WING 2007*, 2007.

[43] Ranjit Jhala and KennethL. McMillan. Array abstractions from proofs. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 193–206. Springer Berlin Heidelberg, 2007.

[44] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 244–256, New York, NY, USA, 1979. ACM.

[45] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[46] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In Stephen M. Watt, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, and Daniela Zaharie, editors, *SYNASC*, page 10. IEEE Computer Society, 2009.

[47] Shuvendu K. Lahiri, Shaz Qadeer, Juan P. Galeotti, Jan W. Voung, and Thomas Wies. Intra-module inference. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 493–508. Springer Berlin Heidelberg, 2009.

[48] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[49] JorgeA. Navas, Peter Schachte, Harald Søndergaard, and PeterJ. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 115–130. Springer Berlin Heidelberg, 2012.

[50] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[51] John C. Reynolds. Automatic computation of data set definitions. In *IFIP Congress (1)*, pages 456–461, 1968.

[52] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[53] Steffen Schlager. *Symbolic execution as a framework for deductive verification of object-oriented programs.* PhD thesis, Karlsruhe Institute of Technology, 2007.

[54] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

[55] Simon Siegler and Nathan Wasser, editors. *Verification, Induction, Termination Analysis: Festschrift for Christoph Walther on the Occasion of His 60th Birthday.* Springer-Verlag, Berlin, Heidelberg, 2010.

[56] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

[57] Christoph Walther. On proving the termination of algorithms by machine. *Artif. Intell.*, 71(1):101–157, November 1994.

[58] Christoph Walther and Stephan Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32(1):35–73, 2004.

[59] Christoph Walther and Nathan Wasser. Fermat, euler, wilson - three case studies in number theory. *Journal of Automated Reasoning*, pages 1–20, 2016.

[60] Nathan Wasser. Generating specifications for recursive methods by abstracting program states. In Xuandong Li, Zhiming Liu, and Wang Yi, editors, *Dependable Software Engineering: Theories, Tools, and Applications — First International Symposium, Nanjing, China*, volume 9409 of *LNCS*, pages 243–257. Springer, 2015.

[61] Nathan Wasser and Richard Bubel. A theorem prover backed approach to array abstraction. In *5th International Workshop on Invariant Generation, held as part of Vienna Summer of Logic, Vienna, Austria*, 2014. Paper available from `https://www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Publications/ALBIA/WING_2014.pdf`.

[62] Nathan Wasser, Reiner Hähnle, and Richard Bubel. Abstract interpretation. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors, *Deductive Software Verification – The KeY Book: From Theory to Practice*, pages 167–189. Springer International Publishing, Cham, 2016.

[63] Benjamin Weiß. *Deductive Verification of Object-Oriented Software — Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, KIT, January 2011.

[64] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.