Localized Static Program Analysis for Optimization and Error Detection

Oskar Haarklou Veileborg

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Localized Static Program Analysis for Optimization and Error Detection

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Oskar Haarklou Veileborg
May 16, 2023

Abstract

Software development processes continuously evolve, but remain imperfect. In this thesis we identify two issues related to software development in the modern programming languages, Java and Go. The first issue concerns a type of software defect commonly known as concurrency bugs. To realize the full potential of modern multi-core processors, programmers must write concurrent, multi-threaded programs. However, development of concurrent programs is complex, and it can be difficult to avoid common shared-memory concurrency bugs, such as data races and deadlocks, in part because scheduling non-determinism makes it hard to reason about behaviors of a multi-threaded program. The Go programming language endorses the use of a channel communication model as a means to avoid concurrency bugs, but recent work shows that Go developers who use channels still introduce many concurrency bugs in their programs, some of which are directly caused by misuse of channels.

The second issue involves abstraction overhead of using functional programming paradigm features in mainstream languages, in particular Java. The use of immutable data and pure functions is an important principle of functional programming. It makes it easier to reason about potential behaviors of programs, which in turn makes it easier to prevent the introduction of bugs. Realizing the benefits of this programming style, mainstream languages have introduced features that are commonly associated with functional programming, such as declarative data-processing. This feature was introduced in Java 8 through the stream API and has been widely adopted. However, previous work shows that using this API instead of imperative data-processing methods comes with a performance cost, referred to as abstraction overhead, which dissuades some developers from using it.

In this thesis we present two novel static program analysis techniques based on abstract interpretation. One enables ahead-of-time optimizations for Java programs that use streams, while the other allows for detection of blocking errors involving channels in Go programs. We built proof-of-concept implementations of these techniques and have experimentally validated their efficacy. The techniques scale to large real-world programs, which consist of both application code and many libraries. Both techniques build on the idea of *localized analysis*, a form of static program analysis that splits the analyzed program into smaller fragments that can be analyzed individually with a high degree of precision.

Resumé

Vores softwareudviklingsprocesser udvikler sig løbende, men de forbliver uperfekte. I denne afhandling identificerer vi to problemstillinger softwareudviklere støder på i de moderne programmeringssprog, Java og Go. Den første problemstilling vedrører en type softwarefejl, der almindeligvis er kendt som concurrency-fejl. For at udnytte det fulde potentiale af moderne multi-core-processorer, skal programmører skrive parallelle, flertrådede programmer. Det er komplekst at udvikle parallelle programmer, og det kan være vanskeligt at undgå almindelige shared-memory concurrency-fejl, såsom data races og deadlocks, bl.a. fordi vilkårligheder i, hvornår hver tråd får lov til at eksekvere, gør det svært at ræsonnere om et flertrådet programs adfærd. Programmeringssproget Go lægger op til brugen af en kanalkommunikationsmodel som et værktøj til at undgå concurrency-fejl, men tidligere forskningsarbejde viser, at Go-udviklere, der bruger kanaler, stadig introducerer mange concurrency-fejl i deres programmer, hvoraf nogle af dem er direkte forårsaget af forkert brug af kanaler.

Den anden problemstilling vedrører det abstraktionsoverhead, der er forbundet med brugen af features fra det funktionelle programmeringsparadigme i mainstream-sprog, især Java. Brugen af uforanderlige data og funktioner uden sideeffekter er et vigtigt princip i funktionel programmering. Det gør det lettere at ræsonnere om programmernes potentielle adfærd, hvilket igen gør det lettere at forhindre, at der opstår fejl. Mainstream-sprogene har erkendt fordelene ved denne programmeringsstil, og har indført features, der almindeligvis forbindes med funktionel programmering, såsom deklarativ databehandling. Denne feature blev indført i Java 8 gennem stream-API'et, og mange udviklere har taget konceptet til sig. Tidligere forskningsarbejde viser imidlertid, at brugen af dette API, fremfor imperative databehandlingsmetoder, er forbundet med en forringet ydeevne, som vi kalder abstraktionsoverhead, hvilket afholder nogle udviklere fra at bruge det.

I denne afhandling præsenterer vi to nye statiske programanalyseteknikker baseret på abstrakt fortolkning. Den ene muliggør ahead-of-time optimeringer for Java-programmer, der bruger streams, mens den anden gør det muligt at opdage concurrency-fejl der involverer kanaler i Go-programmer. Vi har implementeret disse teknikker og har bekræftet deres effektivitet med eksperimenter. Teknikkerne virker selv på store open-source programmer, som består af både applikationskode og mange biblioteker. Begge teknikker bygger på ideen om *lokaliseret analyse*, hvilket er en form for statisk programanalyse, der opdeler det analyserede program i mindre fragmenter, der kan analyseres individuelt med en høj grad af præcision.

Acknowledgments

Let me start by thanking Anders for being a supportive advisor, for the opportunities and experiences I have been granted through my time as a PhD student, both in industry and in academia, and for the great teamwork required to meet rapidly approaching deadlines. I also want to thank Vlad for being an excellent colleague and close collaborator. Getting our Go project shepherded to the finish line took a large combined effort, but it was a good time. Let me also thank Benjamin and Martin for giving me a warm welcome in the research group, and for guidance in my initial years as a PhD student. My colleagues in the PL and LogSem research groups deserve thanks for fostering a very cozy work environment. I want to thank Asger for both friendly rivalry and fruitful cooperation, and for pushing me to be better. Finally, let me thank my family for their continued support, and Camilla for her never-ending encouragement.

Oskar Haarklou Veileborg,
Aarhus, May 16, 2023.

Contents

Part I

Overview

Chapter 1

Introduction

This thesis demonstrates that a *localized* form of static program analysis enables new techniques for optimization and error detection in Java and Go programs. To motivate the need for these techniques, we start with an introduction of software development from a broad perspective.

In our day, software is ubiquitous. Enormous amounts of resources are spent on development, maintenance and execution of software. This thesis presents two pieces of work that can — at a high level — make these processes more efficient.

With some generalization, we can categorize the time software developers spend on construction of software into two categories. The first concerns itself with the addition of features to a product. These features have many different forms, but they are usually aimed at satisfying new needs of users of the product. The second category describes time spent on maintenance in the form of repairing defects and refactoring. Depending on which programming language the software is written in, there are different trade-offs in terms of how easy work in the two categories is (and consequently, how much time it requires). Dynamically typed programming languages typically excel at letting developers get features implemented quickly, while making maintenance and exploration of already written code a bit harder than in statically typed programming languages. In such languages, the type system enforces structure on programs, which both rules out a class of software defects (bugs) known as type errors, and also makes it easier to understand how program executions may unfold up-front. This is not only true for humans, but for machines as well. Statically typed programming languages typically have better tools that enhance the developer experience, such as auto-completion and automated refactoring in Integrated Development Environments (IDE's), and also better analysis tools that can detect bugs and security vulnerabilities in programs. The increased rigidity of programs written in these languages enables tools to automatically reason about them in ways that are currently impossible for programs written in dynamically typed languages. However, both kinds of languages are widely used today.

The 2022 Stack Overflow Developer Survey [64] asked developers which languages they used in the past year, and which languages they love to use. It is clear that

the most used programming languages (as reported in this survey) are dynamically typed, as we find JavaScript and Python in the top of the rankings. However, one third of the respondents report that they used Java in the last year, and more than 16% report that they want to use Go, making it one of the "most-wanted" languages. In the top of the list of loved languages, we find two languages in the functional programming paradigm, Elixir and Clojure, and also Rust and TypeScript, which are statically typed programming languages. The absence of side-effects in programs written in functional programming languages can, like a type system, make programs easier to reason about, which we will discuss later. Rust is an interesting language. It features a type system that is much more complex than what we have seen previously in mainstream programming languages. This system can guarantee the absence of common programming errors involving shared memory, such as use-after-free errors, double-free errors and data races.

Importantly, even when supported by complex type systems, developers still write buggy software. Not only do bugs require developer time to fix, but the bugs themselves can also have severe consequences. Program analysis is a suite of techniques for automatic reasoning about the behavior of programs. Program analysis tools take other programs as input, and attempt to infer some information about how the analyzed program behaves when it is run. This information can be used reason about the presence of bugs, but it can also be used for program optimizations.

Dynamic program analyses execute the analyzed program and attempt to find bugs in the observed executions. Examples include software testing, but also more advanced techniques, such as dynamic taint analysis, that observe whether untrusted (and potentially malicious) inputs flow to security critical parts of the software. Such techniques are limited by only being able to observe a small subset of all possible executions of the program, so they can never prove the absence of errors. In practice, fuzzing tools are used to increase code coverage, and can enable the dynamic analyses to observe an enormous amount of executions. This can increase our confidence in the program's correctness.

Opposite of dynamic program analysis, *static* program analysis techniques reason about programs without running them. They attempt to create a finite model of a program's behavior across all (possibly infinitely many) executions. We can then check whether the model admits undesirable behavior, such as executions that trigger bugs. It is necessary for static analyses to approximate the program's behavior. If the static analysis is *sound*, the inferred model will be an over-approximation. This means that the inferred model captures behavior from *all* possible executions of the program, and potentially also executions that are actually impossible. Traditionally, sound static analysis has been used for program optimization. Having additional information about how a program may behave when it executes enables different forms of optimizations. One of the techniques we propose uses static analysis for this purpose. In terms of error detection, if we determine that none of the modeled executions are buggy, sound program analysis can in principle guarantee the absence of bugs, which is impressive! However, in practice it is very difficult to strike a good balance between efficiency of the analysis and precision, i.e. the degree of over-approximation, which, if severe,

can prevent the analysis from proving absence of errors, even though the program is correct.

Static program analysis tools have been used successfully in the software development industry to find bugs. Notable examples at large technology companies include Infer at Facebook/Meta [14], NullAway at Uber [7], and Tricorder at Google [71]. A long line of research on static program analysis starting in the 1970's paved the way for these achievements.

Concurrency bugs Bugs in one particular category can be extremely difficult to detect, namely concurrency bugs. Modern computers have multi-core processors that can execute many streams of instructions concurrently. To properly exploit the full potential of such processors, the software we write must be concurrent as well. If we can slice up the work that our program needs to perform into independent pieces, the smaller pieces of work can be performed concurrently. Used properly, this approach can lead to large savings in wall-clock execution time. One way to realize this potential is through the use of threads. In a normal, single-threaded, program, the main thread executes instructions from the program's entry-point and onwards, until the program terminates. In a multi-threaded program, threads can choose to spawn new threads with different entry-points. Typically a thread entry-point is a function in the program. All non-terminated threads can then execute concurrently. Modern systems allow programs to have more threads than the processor has cores. In this case, a special program known as the scheduler divides processor time between the different active threads according to some fairness constraints. (Our computers typically run operating systems that group threads in processes. However, most of the intricacies regarding operating systems and processes are irrelevant in our context, so we purposefully do not consider them here.)

In a multi-threaded program, threads typically need to communicate to work together to solve some common task. This communication can happen through reads and writes to shared memory. However, data races are a common issue that can occur when threads communicate through shared memory. A data race occurs when two threads access the same piece of data concurrently, and at least of one the threads wants to modify the data. In this case, there is potential for data corruption, which can lead the program into an inconsistent state. To prevent data races, programmers protect accesses to shared memory with mutual exclusion devices, such as locks. Unfortunately, locks come with their own problems. Deadlocks are a common class of concurrency bugs, where two or more threads end up waiting for events that will never happen. Both data races and bugs involving locks can be frustratingly hard to detect, because triggering them depends on non-deterministic behavior in the scheduler. The huge space of possible execution schedules makes it extremely difficult to comprehend all the possible behaviors of the program, and it makes techniques such as fuzzing less effective.

Inspired by Hoare's communicating sequential processes (CSP), the Go programming language's hallmark concurrency feature is channels. They provide an

alternative way for threads to communicate, compared to traditional communication through shared memory. Although it is claimed that the use of channels prevents many concurrency bugs, recent work shows that developers that write multi-threaded Go applications still face concurrency-related issues, some of which are unique to channels. In this thesis we present a tool that helps Go developers detect channel-related concurrency bugs in their programs.

Abstraction overhead As mentioned earlier, programs written in functional programming languages can be perceived as easier to reason about, compared to programs written in traditional imperative and object-oriented languages. This is mainly due to an emphasis on absence of side-effects. Functional programming's successes have caused typical functional programming concepts to leak into other programming languages. We have new languages that explicitly describe themselves as multi-paradigm, such as Scala. In traditional languages, such as Java, C++, C#, etc., cornerstone concepts of functional programming have been introduced in recent years, such as first-class functions, pattern matching, and declarative data-processing. Developers demand these concepts, and are quick to adopt them, as they enable work at a high level of abstraction, which in turn makes it easier to develop bug-free software. However, we find that some developers explicitly choose not to use these features, because they are associated with an increased performance cost. Previous work shows that the performance cost of using Java's implementation of declarative data-processing is significant in some cases. We call this kind of cost *abstraction overhead*. Due to abstraction overhead, developers must sometimes choose between performance and comfortability.

In Java, source programs are initially compiled to bytecode. Bytecode is a simplified program representation that is suitable for interpretation by a virtual machine (VM). Almost all program optimizations performed in Java happen through the just-in-time (JIT) compiler, which is embedded in the VM. As the VM interprets bytecode, it simultaneously collects information about the execution that is relevant to optimizations, such as run-time type information. When enough information has been collected, and some performance-critical parts of the program have been identified, the JIT compiler uses the gathered information to optimize the interpreted bytecode into natively executable machine code. Gathering information while the program executes and optimizing bytecode uses precious execution time, so the JIT compiler has to favor simple optimizations over complex ones.

Ahead-of-time (AOT) optimization techniques have been effective for languages that compile directly to native machine code, such as C, C++, and languages in the ML family. These techniques have the advantage that the time spent on optimizations is separate from the time spent executing the program, so they can afford to apply analyses that are too expensive to perform in a JIT setting. Additionally, the optimizer can tell the developer if optimizations have been applied successfully. The disadvantage of these techniques is that they do not have access to run-time information, which makes the application of some optimizations harder. To realize the full performance

potential of Java programs, it may be time to pursue AOT optimization techniques for bytecode. However, strong encapsulation guarantees in the Java language makes traditional optimization techniques non-trivial to apply. In this thesis we present an AOT bytecode optimization technique for Java aimed at removing the abstraction overhead of using declarative data-processing.

In the next sections, we summarize the issues presented here that developers face when they develop modern software, as well as contributions we have made towards alleviating them, particularly through precise and scalable static analysis for programs written in statically typed programming languages. We show how to achieve a good trade-off between precision and scalability by employing *localized* static analysis, which splits a program into smaller fragments that can be analyzed independently.

1.1 Research Challenges

Based on the issues presented in the introduction, we describe our four main challenges that we investigate in this thesis:

C1 Declarative data-processing with streams is a popular functional-style feature introduced to Java in version 8. Developers want to use functional abstractions in their programs, as they make programs easier to comprehend, and make it easier to avoid some classes of bugs. However, abstraction overhead persuades some developers to avoid the abstractions due to a performance cost. Can we design a technique that eliminates the need for such a choice?

How can we obtain the benefits of a more functional programming style in modern Java, in particular declarative data-processing, without a performance penalty?

C2 The program optimizations performed in Java are almost fully restricted to the JIT compiler. However, AOT optimizations may have their own benefits. Java's strong encapsulation guarantees makes some traditional AOT optimizations difficult to perform at the bytecode level. Can we overcome these difficulties and bring the benefits of AOT optimizations to Java?

How do we realize the benefits of AOT optimization techniques in languages that are traditionally optimized at run-time?

C3 The creators of the Go programming language encourage developers to use channels as a means of communication between threads, as opposed to traditional shared-memory communication. The rationale is that the use of channels prevents many classical concurrency-related bugs. However, studies show that Go programs contain many *blocking errors*, some of which are directly related to channels. Can we design a technique that can effectively and statically detect such channel-related bugs?

How can Go programmers detect and avoid concurrency bugs that are common to channel-related communication?

C4 It can be very difficult to strike a good balance between static analysis performance and precision, especially for analyses involving pointers (described in section 2.3). Designing highly precise pointer analyses that scale to large real-world code bases, which consist of both application code and many libraries, is an active area of research. We find that solutions to the previous challenges (C1-C3) based on static analysis techniques need precise handling of pointers. How can we circumvent the scalability issues that typical whole-program pointer analyses face?

How can we design extremely precise, goal-oriented, program analyses that still scale to large real-world Java and Go programs?

1.2 Contributions

The thesis is based on research papers that present solutions to the previously mentioned challenges. They are included as published in Part II with minor visual changes to accommodate differences between page layouts. Additionally, some sections are included that describe unpublished follow-up work relevant to the techniques presented in the publications. These sections are written solely by the author, and their titles are prefixed by *Epilogue*.

P1 *Eliminating Abstraction Overhead of Java Stream Pipelines using Ahead-of-Time Program Optimization*

Anders Møller and Oskar Haarklou Veileborg. Published in the Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA, November 2020. Included in Chapter 4.

P2 *Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation*

Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Published in the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 2022. The paper received the ACM SIGSOFT Distinguished Paper Award. Included in Chapter 5.

The above papers describe novel analysis techniques that improve state of the art for the four challenges outlined in section 1.1. For challenges C1 and C2, we present an AOT program transformation tool for Java bytecode which optimizes programs that use streams for declarative data-processing (P1, Chapter 4, [56]). We empirically evaluate our technique through a proof-of-concept implementation, and find that our optimizations can remove the abstraction overhead of using streams in most cases. The program transformations are guided by a novel static analysis technique that is tailored to very precise analysis of stream pipelines. Furthermore, the technique is API-agnostic, and we show that it enables optimizations for two different kinds of stream API implementations. We also show how bytecode optimizations can be

tricky due to strong encapsulation. However, being able to apply inlining and stack-allocation transformations aggressively, and selective application of code cloning, allows the optimizer to produce bytecode that does not violate Java's encapsulation rules.

For challenge C3, we propose a novel static analysis technique for detection of blocking errors involving channels (P2, Chapter 5, [86]). Through pragmatic design choices, the static analysis can model interesting executions of multi-threaded programs, without succumbing to the problem commonly known as *combinatorial explosion*. We empirically evaluate our technique through a proof-of-concept implementation, and find that it is able to detect more blocking errors than previous state-of-the-art techniques on a suite of 6 large real-world Go programs. It also has an acceptable ratio of true to false reports.

Towards challenge C4, we show how localized analysis (analysis of small program fragments) enables us to perform analyses that are typically too expensive to run for whole programs. The static analysis techniques proposed in both publications build on this idea, and demonstrate that it is very effective. Restricting the analysis to selected parts of a program requires careful considerations to preserve soundness. In the static analysis technique developed for P1, the program fragment to be analyzed is not determined up-front, but is computed based on how interesting objects flow through the program. The analysis soundly skips calls to functions that cannot impact such objects. In P2 we propose a static analysis technique that analyzes fragments that are constructed before the analysis starts. Through the use of a pre-analysis that over-approximates side-effects of function calls, the static analysis can model the effects of function calls that go outside the defined fragment. Splitting a program into fragments that can be analyzed separately is not a novel idea, but we show that it is a surprisingly powerful and versatile technique, and we show how it can be applied in two apparently different situations.

The author of this thesis has made major contributions to the processes that lead to the publication of the included papers. This includes ideas, tool design and implementation, experimental evaluation and paper writing. The optimization tool developed for the first paper, STREAMLINER, was developed solely by the author, while the static analysis tool for blocking error detection developed for the second paper, GOAT, was developed in part by co-author Georgian-Vlad Saioc. Both tools are open-source and are available at `brics.dk/streamliner` and `brics.dk/goat`, respectively.

1.3 Outline

The thesis is structured into two parts. Part I contains an overview of the research area surrounding the author's work, while published research papers are included in Part II.

Part I begins with this introduction. Then, Chapter 2 presents fundamental background material on static program analysis, such as lattices and monotone frameworks. The theory and practices presented there are the pillars upon which our novel static

analysis techniques are built. In Chapter 3 we present the concept of declarative data-processing. First through the use of lists, but then through use of streams, which is an abstraction that has been implemented in mainstream programming languages, in two different styles. The content of this chapter serves as background material for the paper included in Chapter 4. Part II contains published research papers that present solutions to our challenges, and they have individual conclusions.

Chapter 2

Program Analysis

This chapter aims to introduce background theory of program analysis, which is relevant for both publications (Chapters 4 and 5) presented in the thesis. Specifically we will look at the branch of program analysis known as *static program analysis*. Static program analysis is a collection of techniques that aim to reason mathematically about the behavior of a program without running it. This is different from *dynamic* program analysis, which mainly reasons about programs by inspecting concrete executions of the analyzed program. The contents of this chapter is based on previous work by Cousot and Cousot [19, 20], Kam and Ullman [35] and Kildall [38].

The analyses we develop aim to tell us some interesting things about the programs that are analyzed. If we want to perform automatic program optimization, static analyses may be employed to figure out when it is safe to apply the optimization. In the case of bug detection, we use static analysis to figure out if the program has undesirable (buggy) executions. Unfortunately, Rice's Theorem [69] states that determining whether a program's behavior satisfies a "non-trivial" property is, in general, undecidable. In practice, this has the consequence that automated techniques which reason about properties of behavior of programs must approximate in one of two directions. Techniques can over-approximate the behavior of a program, modeling everything that the program can do when it is run, at the cost of possibly also modeling behavior that cannot occur. Over-approximation of behavior gives a *sound* but *incomplete* analysis. A program's behavior can also be under-approximated. Here the model of the program's behavior is known to only include behavior that can actually occur when it is executed, at the cost of not capturing the behavior of all possible executions of the program. This kind of approximation gives a *complete* but *unsound* analysis. The following sections introduce some program analysis fundamentals.

2.1 Lattice Theory

Lattices are ubiquitous in program analysis. They offer a convenient way to consolidate information collected by the analysis along different execution paths in an

(a) Channel status lattice used in Chapter 5. (b) Sign lattice used for Java bytecode cleanup in Chapter 4.

Figure 2.1: Examples of lattices used in later chapters.

analyzed program. Their use also enable neat proofs of desirable properties of program analyses, such as soundness and completeness.

A lattice is defined in terms of a set of elements A and a partial order $\sqsubseteq$ on the elements in A. An upper bound a for a set of elements $S \subseteq A$ satisfies $\forall s \in S : s \sqsubseteq a$, written simply as $S \sqsubseteq a$. A *least* upper bound for a set of elements S, written $\bigsqcup S$, is an upper bound for S, and is less than any other upper bound of S: $S \sqsubseteq a \Rightarrow \bigsqcup S \sqsubseteq a$. The least upper bound $\bigsqcup S$ is also referred to as the *join* of the values in S, and we will often write $x \sqcup y$ instead of $\bigsqcup\{x, y\}$. We have analogous definitions for (greatest) upper bounds, where the dual operator $\bigsqcap S$ is referred to as the *meet* of the values in S. The pair of A and $\sqsubseteq$ defines a lattice if $x \sqcup y$ exists for all $x, y \in A$, and a lattice is said to be complete if $\bigsqcup S$ exists for all $S \subseteq A$. We will only be working with complete lattices in this thesis.

Lattices are often illustrated as Hasse diagrams, where the least elements are in the bottom and the greatest are in the top, and the partial order is depicted with lines between elements. Two examples of lattices are shown in fig. 2.1. Elements of the first lattice are used to represent the status flag of channels in Go, which can be either *open* or *closed*. The elements $\top$ and $\bot$ represent *unknown* and *undefined*, respectively. The result of joining the *OPEN* and *CLOSED* elements is $\top$. Elements of the second lattice, also known as the *Sign* lattice, represent integers that are negative (-), zero, or positive (+). The top element in this lattice represents all integers, and is therefore an upper bound of any subset of the other elements. When we use lattices in program analysis, the elements of the lattice are usually abstract representations of some set of concrete items from the analyzed program. In the channel status lattice, the elements abstractly represent sets of boolean values, while the elements in the *Sign* lattice represent sets of integers. However, we will also see that lattices are used to represent sets of more complex structures, such as program heaps and complete program states. We relate lattice elements and concrete items with abstraction and concretization functions, denoted as α and γ, respectively [19]. Using the *Sign* lattice as an example, the signatures of these functions are given as $\alpha : \mathit{Sign} \to \mathscr{P}(\mathbb{Z})$ and $\gamma : \mathscr{P}(\mathbb{Z}) \to \mathit{Sign}$. With the set-based view on lattice elements, we say that a lattice element x is more *precise* than y when $x \sqsubset y$.

The height of a lattice is defined as the length of the longest chain of unique elements satisfying $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \cdots$, where $a_i \in A$. Both lattices in fig. 2.1 have height 3. Lattices with finite height are of special interest, as we will see in section 2.2.

Standard lattice constructions There are two standard lattice constructions that are used throughout the publications in this thesis: product lattices and map lattices. A product lattice on two sub-lattices, $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$, has elements in the Cartesian product $A \times B$ (hence the name), and the partial order is defined componentwise:

$$(a_1, b_1) \sqsubseteq (a_2, b_2) \iff a_1 \sqsubseteq_A a_2 \wedge b_1 \sqsubseteq_B b_2$$

It follows that least upper bounds and greatest lower bounds can be decomposed componentwise as well. Product lattice elements are conveniently used to represent values with multiple independent components. For instance, we use a product lattice with sub-lattices for different concrete Go types to abstractly represent the set of *all* concrete values in Go in Chapter 5.

The other standard lattice construction is a map lattice, which is defined in terms of a set X and a lattice $(A, \sqsubseteq_A)$, and is usually written as $X \to A$. The elements of $X \to A$ are functions that map elements in X to elements in A. We write $[x_1 \mapsto a_1, x_2 \mapsto a_2, \ldots]$ for the function that maps x_1 to a_1, x_2 to a_2, etc. The partial order for map lattices is defined in terms of a pointwise decomposition:

$$\forall f, g \in X \to A : f \sqsubseteq g \iff \forall x \in X : f(x) \sqsubseteq_A g(x)$$

And, similarly to product lattices, least upper bounds and greatest upper bounds can also be decomposed pointwise on each element of X. In program analysis we typically use map lattice elements to represent values for a set of program variables. For instance, if *Vars* denotes the set of variables in the analyzed program, the lattice $\mathit{Vars} \to \mathit{Sign}$ could be used to abstractly represent sets of integers for all variables. Another typical use of map lattices is to distinguish state at different program points. In this case we will usually refer to the domain as a set *Nodes* of control locations. This idea can also be lifted further to distinguish states based on more properties of an execution, such as calling contexts.

2.2 Monotone Frameworks

Monotone frameworks are a systematic way to design program analyses, which gives guarantees about decidability, precision and correctness. The main idea is to derive a system of equations from a control flow graph (CFG) representation of a program, which is subsequently solved methodically. The nodes in a control flow graph are associated with instructions in the source program, and directed edges between nodes represent possible control flow. Most nodes have a single successor, but some nodes associated with branching, function call and return instructions can have multiple successors. We associate with each control flow graph node v a constraint variable $[\![v]\!]$, which holds an element of some complete analysis lattice L with finite height. Typically the elements of this lattice are abstract representations of program state for a single program point. The element $[\![v]\!]$ describes the state of the program in the program model computed by the analysis *after* the instruction associated with v is

executed.[1] This gives a so-called *flow-sensitive* analysis, which is an analysis where the control flow between instructions is taken into account. The analyses presented in both included publications are also flow-sensitive, as they are instantiations of monotone frameworks.

For each node v, we emit an equation: $[\![v]\!] = f_v(x)$. The function f_v is the *constraint function* associated with v, and it describes how the program's state after executing the instruction for v is related to the program's state at other points in the program. Here x is an element of the lattice: $Nodes \to L$, which allows f_v to access the state at every point in the program, but typically only the program's state at v's neighbors is relevant. We can then gather the constraints into a unified analysis constraint function: $f(x) = [v \mapsto f_v(x) \mid v \in Nodes]$. We seek a solution to the constraint system, i.e. an assignment, x, of values to all $[\![v]\!]$, that satisfies the constraints.[2] Such an assignment is a fixed point of the analysis constraint function: $x = f(x)$. While any fixed point is a solution, we are interested in the *least* fixed point, which is the most precise solution to our constraints.

It is imperative that the constraint functions are monotone. For a lattice L, a function $f : L \to L$ is monotone only if $\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. When the constraint functions f_v are monotone, the unified analysis constraint function is monotone as well. Due to Kleene's fixed-point theorem [40], we know that the unique least fixed point of such a function is defined as follows:

$$lfp(f) = \bigsqcup_{i \geq 0} f^i(\bot)$$

A consequence of monotonicity of f is that $\bigsqcup_{0 \leq i \leq k} f^i(\bot) = f^k(\bot)$. Because L has finite height, there exists a k such that the increasing chain

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots \sqsubseteq f^k(\bot) \sqsubseteq f^{k+1}(\bot) \sqsubseteq \cdots$$

stagnates at $f^k(\bot)$, i.e. $f^k(\bot) = f^{k+1}(\bot)$. This gives us an algorithm for computing the most precise solution to our constraint system. We can start with the bottom element of the lattice $Nodes \to L$ and iteratively apply the function f until the result does not change. This is known as the "naive fixed-point algorithm".

Imagine that we want to design an analysis that can reason about potential nullability of variables at run-time. For each variable and program point, we want to know if that variable is guaranteed to contain a non-null reference, or if it may contain the null pointer. For this purpose we may use the lattice $States = Vars \to Null$ to represent abstract states. Each abstract state associates with each variable an element of the lattice *Null*, which is a simple 2-element lattice with a $\top$ and $\bot$ element. $\bot$ abstractly represents values that are guaranteed to be non-null, while $\top$ represents all values (including null).

[1] In a backwards analysis (described later), the element describes the state of the program *before* the instruction has been executed.

[2] $[\![v]\!]$ and $x(v)$ are used interchangeably.

Here we give constraint functions for a CFG node v representing an assignment to a variable X, depending on the right-hand side of the assignment.

$$\begin{aligned} X \texttt{ = null:} \quad & [\![v]\!] = \mathit{wrap}(\lambda\sigma.\,\sigma[X \mapsto \top]) \\ X \texttt{ = alloc T:} \quad & [\![v]\!] = \mathit{wrap}(\lambda\sigma.\,\sigma[X \mapsto \bot]) \\ X \texttt{ = } Y\texttt{:} \quad & [\![v]\!] = \mathit{wrap}(\lambda\sigma.\,\sigma[X \mapsto \sigma(Y)]) \\ & \mathit{wrap}(t_v) = t_v\left(\bigsqcup_{w \in \mathrm{pred}(v)} [\![w]\!]\right) \end{aligned}$$

The *wrap* function computes the abstract state just prior to the execution of the instruction associated with v and passes it on to the *transfer function*. We use t_v to refer to the transfer function associated with v. The prior abstract state is computed by taking the join of abstract states at all control flow predecessors of v. The transfer functions then return an updated abstract state where the effect of executing the associated instruction is reflected. For assignments that directly assign null or a reference to a newly allocated object, the abstract value for X is simply updated to $\top$ or $\bot$, respectively.[3] However, for an assignment with a right-hand side that is another variable Y, the abstract value for X becomes the same as the abstract value for Y, which represents the flow of a value from Y to X.

It is convenient to use transfer functions as shown in the example, as it lets us focus on how an abstract state is transformed based on the executed instruction. Constraint functions in many analyses can be phrased in terms of transfer functions, where we only need to describe how instructions affect the abstract state at the point before the instruction is executed. Analyses that can be phrased in this way can have their constraint systems solved more efficiently than with the naive fixed-point algorithm. In each iteration of the naive algorithm, all of the constraint functions are run. However, we do not expect all of the constraint variables to obtain new values in each iteration. The constraint functions presented above only "return something new" when the abstract state at one of the predecessors changed in the previous iteration. This observation leads to the design of the *work-list fixed-point algorithm*, which is outlined in algorithm 1. It works by keeping track of a work-list of CFG nodes that need to have their transfer functions re-run. Initially the work-list contains all nodes. In each iteration of the loop, a node v is removed from the work-list and the value of $[\![v]\!]$ is recomputed. If the value changes, we add to the work-list the nodes whose constraint functions depend on the value of $[\![v]\!]$, i.e. the successors of v. In this way we can avoid a lot of redundant work compared to the naive fixed-point algorithm.

Variants of the work-list algorithm are used in practice to solve constraint systems in monotone frameworks, and it is used in the implementations developed for both publications in the thesis. The definition does not impose an order in which the nodes should be picked from the work-list. However, different orders can make a

[3] The syntax $x[y \mapsto z]$ denotes a function that returns z when given y, and returns the same as x on all other inputs.

Algorithm 1: Work-list algorithm

$x = [v \mapsto \bot \mid v \in \mathit{Nodes}]$
$W = \mathit{Nodes}$
while $W \neq \emptyset$ **do**
  $v = \texttt{Pop}(W)$
  $y = t_v\left(\bigsqcup_{w \in \text{pred}(v)} x(w)\right)$
  if $x(v) \neq y$ **then**
    $x = x[v \mapsto y]$
    $W = W \cup \text{succ}(v)$

large difference on the performance of the algorithm in practice, and choosing a good one has been a subject of previous work [18, 34]. The work-list algorithm used in Chapter 4 orders nodes intraprocedurally (in the same function) in a FIFO (first-in-first-out) fashion, modeling a queue, while nodes that are in different functions are ordered in a LIFO fashion. Because the analysis is infinitely context sensitive (see below), the order that functions are explored in corresponds to a depth-first exploration of the call tree. The work-list algorithm used in Chapter 5 orders nodes in a reverse post-order based on the structure of the interprocedural control flow graph.

Variations We can classify standard instantiations of monotone frameworks according to different parameters. One parameter is whether the analysis is a *forward* or *backward* analysis. The analysis presented earlier is a forward analysis, and is characterized by transfer functions that transforms abstract states from control flow predecessors into an abstract state where the effect of executing an instruction is reflected. In a forward analysis, the constraint function for a node depends on constraint variables in predecessors nodes. This is important in the implementation of the work-list algorithm, where we add the inverse of this dependency relation (the successors) to the work-list when a constraint variable changes. In a backwards analysis, each transfer function derives some information about the program state *before* the instruction is executed. It derives this based on program state from "the future". A typical example of a backwards analysis is live variables analysis. A variable is live at a program point if, in an extension of execution from this point, the variable is read before it is assigned. This property can be approximated by a program analysis. In this analysis, the transfer functions operate on elements of a powerset lattice that represent the set of program variables that may be live. For an assignment to a variable X, the set of variables that are live before the instruction is those that are live just after the instruction, *except for X* (because we are assigning it just now), and the variables that are used in the right-hand side of the assignment. In a backwards analysis, the input for transfer functions is computed as the join of abstract states at all control flow successors of the current node, as opposed to the predecessors, and the dependency relation (relevant to the work-list algorithm) is also inverted.

In broad strokes, forward analyses are useful for approximating information that holds for executions up until each instruction, while backwards analyses approximate information that holds in the future for executions starting from each instruction.

Another common characteristic that differentiates analyses is whether they are *context-sensitive*. In the monotone framework, a context-sensitive analysis infers an abstract program state not just for each CFG node, but for each combination of node and context. Usually contexts are derived from some property of the execution that lead to a function being called, which can relieve some of the imprecision that arises when information from different call sites of the same function is joined. Imagine a scenario where we perform sign analysis on a program that uses the identity function in different places. At one call site the function is called with a positive argument, while it is called with a negative argument at another call site. If we are not able to separate the abstract states in these two calls, the abstract element representing the parameter of the identity function will necessarily become imprecise, as we will join the abstract elements for the positive and negative parameters. For the identity function, this imprecision carries over to the returned value, and this can have downstream consequences for the precision of the rest of the analysis.

A common choice of context is *k-limited call strings*, where a context contains the call sites of the top k functions in an abstract call stack. This choice of contexts allows us to avoid imprecision due to joined information from different callers for call trees of depth up to k. The downside of context-sensitive analysis is that the analysis can become more expensive to compute, because we need to infer abstract information for more program points. The full analysis lattice (in a monotone framework) goes from $\mathit{Nodes} \to L$ to $\mathit{Contexts} \times \mathit{Nodes} \to L$, so the size of the set *Contexts* is significant for performance. Luckily, it is unexpected that all possible contexts are realizable in a program, as there is usually some structure to the way that functions call each-other. The choice of domain of contexts presents trade-offs, as you want to strike a balance between performance and precision. This choice is further complicated by the fact that increased precision can make some analyses terminate faster.

2.3 Pointer Analysis

Pointer analysis is a very common analysis to perform for programs written in languages that support dynamic memory allocation and references as first-class values. The goal of pointer analysis is to answer questions of the form: "which pieces of memory can some variable point to at run-time?" When analyzing programs written in object-oriented programming languages, it is common to phrase the question in terms of objects, i.e. which objects can some variable point to. Being able to answer such questions gives insights about the overall structure of the program. Also, for languages that support dynamic calls (perhaps through dynamic dispatch in object-oriented languages, or through function pointers), precise pointer analysis is required to be able to statically resolve such calls. Because pointer analysis results usually depend on the call graph, we often compute call graphs simultaneously with pointer

analyses, referred to as *on-the-fly* call graph construction.

An infinite program execution can allocate an unbounded amount of concrete memory cells. We cannot hope to distinguish all of the allocations, as our analysis abstractions must be finite. A common choice is to introduce one abstract cell per syntactic allocation site, which will abstractly represent all concrete cells allocated at that instruction. This is known as the *allocation-site abstraction* [16]. It is versatile in the sense that the abstraction can be enriched in various ways. We can choose to not only identify cells by syntactic allocation site, but by the combination of allocation site and calling context. We use this variant in Chapter 4, and it is commonly known as heap cloning or context-sensitive heap [60, 76]. In Chapter 5 we use a variant of allocation-site abstraction that additionally distinguishes cells based on the thread that allocated memory.

Because an abstract allocation site may represent multiple concrete cells, modeling memory writes often has to done *weakly*. If we have an abstract heap $\sigma : \mathit{Cells} \to L$, an abstract cell $o \in \mathit{Cells}$, and we want to model the effect of an instruction that assigns an abstract value $x \in L$ to o, we must use the constraint $\sigma(o) \sqsupseteq x$ (or the equivalent: $\sigma(o) = \sigma(o) \sqcup x$) instead of $\sigma(o) = x$. In a concrete execution, the instruction writes to exactly one of the cells that are abstracted by o, but in the analysis o may abstract multiple concrete cells. We must soundly model that some cells represented by o retain their original value, which is done by stating that cells represented by o contain a value that is the join of the previous value and x. This can be unsatisfactory for precision, so in the analyses we developed for the included publications, we use a technique known as *strong updating* [16]. When the analysis can prove that an abstract memory cell exactly corresponds to a single concrete memory cell, we can soundly use the precise model of memory writes.

Pointer analysis can be framed in the monotone framework, but for large programs, flow-sensitive pointer analysis is typically too expensive. Instead, flow-insensitive pointer analyses are used, such as variants of Steensgaard's pointer analysis [80] and Andersen's pointer analysis [3] with different degrees of context-sensitivity. Designing pointer analyses that strike a good balance between precision and performance is still an active area of research. Both approaches described in the included publications rely on whole-program pointer analysis as a pre-analysis, and we find that state-of-the-art pointer analysis implementations are still inadequate in terms of precision and performance.

Chapter 3

Declarative Data-processing with Streams

There are many ways to define declarative data-processing, streams, and their combination. This chapter aims to give a common understanding of these concepts, particularly in relation to the way they are used the first publication (Chapter 4).

Declarative data-processing is a concept that is typically attributed to functional programming languages. Abstractly, it is a style of data transformation where you aim to describe the properties of the output given the input, and defer the task of specifying exactly the intermediate steps the program should execute to derive this input, to the compiler and/or data-processing library. An example of declarative data-processing is given in fig. 3.1, where a piece of OCaml code computes the sum of some even squares. Although this computation is slightly contrived, it is a recurring example in Chapter 4 and related work. The main point to notice is that all the transformations of the input data (the list of numbers from 0 to 9) happen via calls to higher-order functions in the `List` standard library. For instance, in the call to `List.map (x -> x * x)`, we specify that in the resulting list, each element should be the squared version of the element in the input list. Notably we do not instruct OCaml in how the list is actually constructed or write any code to traverse the input list. This is all hidden inside the library. Declarative data-processing works at a high level of

```
1 (* Construct a list of the numbers from 0 to 9 *)
2 List.init 10 (fun x -> x)
3 (* Square the numbers in the list: [0, 1, 4, 9, ..., 81] *)
4 |> List.map (fun x -> x * x)
5 (* Restrict to even numbers: [0, 4, 16, 36, 64] *)
6 |> List.filter (fun x -> x mod 2 == 0)
7 (* Sum the numbers in the list and print the result: 120 *)
8 |> List.fold_left (+) 0 |> print_int
```

Figure 3.1: Example of computing the sum of some even squares in OCaml.

```
1 (* Construct a stream of the numbers from 0 to 9 *)
2 Seq.init 10 (fun x -> x)
3 (* Transform the stream: elements become squared *)
4 |> Seq.map (fun x -> x * x)
5 (* Transform the stream: odd elements are discarded *)
6 |> Seq.filter (fun x -> x mod 2 == 0)
7 (* Sum the numbers in the stream and print the result: 120 *)
8 |> Seq.fold_left (+) 0 |> print_int
```

Figure 3.2: Computing the sum of some even squares with streams.

abstraction, and has the advantage that the intended behavior of the program is often clear, compared to if the program was implemented in an imperative style, where the intended behavior must be derived from the intermediate steps describing how to compute the output from the input.

One disadvantage of the implementation in fig. 3.1 and list-based data transformations in general is that the transformations are applied eagerly (we assume a programming language with strict, i.e. eager, evaluation semantics common to most modern programming languages). The intermediate operations that output new lists (`List.map` and `List.filter` in the example) consume and transform the full input list before control is passed on to the next operation. For instance, the `map` operation creates a new temporary list that is processed once by the `filter` operation and is then thrown away. This is wasteful both in terms of time and space. There are some scenarios where the overhead is more pronounced: A common operation to perform is to retrieve the first n elements of the output list (where n is much smaller than the input) and discard the rest. In this case, naively applying the element-wise intermediate transformations (for instance the squaring operation passed to `List.map` in the example) for elements that will not be part of the output is wasteful, especially if the operations themselves are expensive to perform. Previous work has addressed these issues for pure programs amenable to equational reasoning through *listlessness* [88, 89], *deforestation* [30, 90] and *stream fusion* [22, 39]. However, these techniques cannot be transferred directly to impure programs (which most programs written in modern programming languages are). To fix the problem in general we need to move away from traditional data transformations based on temporary lists and use *streams* instead.

Streams (also known as stream pipelines, lazy lists, delayed lists and functional iterators) consist of a stream source, some intermediate operations that transform the elements of the stream, and a terminal operation that consumes the elements and produces the desired result.[1] The same example as in fig. 3.1 is implemented with streams in fig. 3.2. The OCaml standard library `Seq` for streams has a similar interface as the `List` library, so the code looks near identical to the original example. While the end result of running both programs is the same, the evaluation order of

[1] Some definitions of streams allow reuse of sources and intermediate streams with multiple terminal operations, but we do not consider this to be a requirement.

the anonymous functions passed to `init`, `map` and `filter` is different, which would be noticeable if they had side effects. In the list-based code, all of the calls to the `init` anonymous function execute before the calls to the `map` anonymous function, which in turn execute before the calls to the `filter` anonymous function. In the code with streams, the calls are inter-leaved. First, a call to the `init` anonymous function happens, which returns a number, this number is then passed through the `map` anonymous function and is replaced with its square. Afterwards the number is passed to the `filter` predicate, and if the number passes the test, the number finally enters the terminal left fold operation. If the stream was not exhausted, the process starts over with the next element of the stream source. This design avoids allocation of intermediate temporary lists and allows for potential early termination in terminal operations without exhausting the stream source, which is useful for the operation mentioned above of returning the first *n* elements (a terminal operation that can return without exhausting the stream source is also referred to as a *short-circuiting operation*). In essence, streams allow programmers to realize the benefits of the combination of declarative data-processing and lazy evaluation semantics in mainstream programming languages.

3.1 Push- & Pull-style Stream Implementations

There are two main flavors to choose from when implementing the stream abstraction, referred to as *pull-* and *push-style*, respectively. When we work with full stream pipelines consisting of a source, some intermediate operations, and a terminal operation, i.e. when we are users of an already implemented abstraction, it is generally not possible to tell the difference. The difference lies in the type (or API) of the stream objects themselves and the control flow of the execution of a stream pipeline. Figure 3.3a contains two example OCaml types defining streams in push- and pull-style.[2] The push-style stream interface accepts a consumer function that will be called by the stream object for each element in the stream. The returned boolean flag signals to the stream source whether more elements can be consumed, which is useful for the implementation of short-circuiting operations. In this style, control is handed over to the stream object until the stream is exhausted or early termination is signaled. The pull-style stream interface is a function that can be called to retrieve the next element of the stream (if any). This strongly resembles iterators in conventional programming languages.

Most common stream operations can be implemented in both styles, including `map`, `filter`, `concat` (ordered concatenation of two streams), `flatMap` (given a function that transforms an element to a stream, return the concatenation of the streams produced by applying the function to all elements in the original stream), and the left

[2] Of course many variations are possible. These types make it necessary for implementations to be impure, which can be avoided with different definitions, but this (push-style stream) definition closely resembles the type of streams in Java and other object-oriented programming languages with a stream abstraction.

```
type 'a pushstream = ('a -> bool) -> unit
type 'a pullstream = unit -> 'a option
```

(a) Type definitions for push- and pull-style streams.

```
let list_to_pushstream (l: 'a list): 'a pushstream =
    fun (consume: ('a -> bool)) ->
        let rec aux = function
            | x :: xs -> if consume x then aux xs
            | [] -> ()
        in aux l

let map_pushstream (f: 'a -> 'b) (s: 'a pushstream): 'b pushstream =
    fun (consume: ('b -> bool)) ->
        s (fun (x: 'a) -> consume (f x))

let filter_pushstream (p: 'a -> bool) (s: 'a pushstream): 'a pushstream =
    fun (consume: ('a -> bool)) ->
        s (fun (x: 'a) -> if p x then consume x else true)

let sum_pushstream (s: int pushstream): int =
    let acc = ref 0 in
    s (fun (x: int) -> acc := !acc + x; true);
    !acc

let list_to_pullstream (l: 'a list): 'a pullstream =
    let r = ref l in
    fun () ->
        match !r with
        | x :: xs -> (r := xs; Some x)
        | [] -> None

let map_pullstream (f: 'a -> 'b) (s: 'a pullstream): 'b pullstream =
    fun () ->
        match s () with
        | Some x -> Some (f x)
        | None -> None

let filter_pullstream (p: 'a -> bool) (s: 'a pullstream): 'a pullstream =
    let rec aux () =
        match s () with
        | Some x -> if p x then Some x else aux ()
        | None -> None
    in aux

let sum_pullstream (s: int pullstream): int =
    let rec aux acc =
        match s () with
        | Some x -> aux (acc + x)
        | None -> acc
    in aux 0
```

(b) Map, filter & sum implementations for both stream definitions.

Figure 3.3: Example OCaml implementation of a stream abstraction in both styles.

fold terminal operation. Example implementations for the `filter`, `map`, and `sum` (a specialization of left fold) operations are given for both styles in fig. 3.3b. While they can provide much of the same functionality, there are advantages and disadvantages to both styles. It is not possible to implement the `zip` operation, which unifies two streams into a stream of pairs, on infinite streams with a push-style abstraction. The reason is that control has to be transferred to one of the streams until it is exhausted, which means that we will never get an opportunity to traverse the second stream. With a pull-style stream, control is passed back to the `zip` operation after a single element from the first stream is found, so it is possible to then request the next element of the second stream, and then pass both elements on as a pair. However, push-style streams have the advantage that the execution of a stream pipeline generally only contains a single *loop* (barring the behavior of user-provided functions), which is located in the stream source. Contrast this with pull-style streams, where we find loops (modeled as recursive functions in OCaml) in both the left fold operation and the `filter` operation. The presence of only a single loop can make execution of the stream easier to optimize for just-in-time compilers such as the HotSpot compiler found in the Java virtual machine (JVM). Java's stream abstraction is implemented in push-style, while equivalent abstractions such as views in Scala and Language-Integrated Queries in C# are pull-style.

Although a stream is a nice abstraction for declarative data-processing, we will see that its use, particularly in Java, comes with a performance cost when compared to imperative-style implementations. In Chapter 4 we explore the cause of the so-called *abstraction overhead* in Java's stream implementation, and develop an ahead-of-time optimization technique that mitigates the overhead for both push- and pull-style stream implementations.

Part II

Publications

Chapter 4

Eliminating Abstraction Overhead of Java Stream Pipelines using Ahead-of-Time Program Optimization

By Anders Møller and Oskar Haarklou Veileborg. Published in the Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA, November 2020. Section 4.11 is new material.

Abstract

Java 8 introduced streams that allow developers to work with collections of data using functional-style operations. Streams are often used in pipelines of operations for processing the data elements, which leads to concise and elegant program code. However, the declarative data processing style comes at a cost. Compared to processing the data with traditional imperative language mechanisms, constructing stream pipelines requires extra heap objects and virtual method calls, which often results in significant run-time overheads.

In this work we investigate how to mitigate these overheads to enable processing data in the declarative style without sacrificing performance. We argue that ahead-of-time bytecode-to-bytecode transformation is a suitable approach to optimization of stream pipelines, and we present a static analysis that is designed to guide such transformations. Experimental results show a significant performance gain, and that the technique works for realistic stream pipelines. For 10 of 11 micro-benchmarks, the optimizer is able to produce bytecode that is as effective as hand-written imperative-style code. Additionally, 77% of 6 879 stream pipelines found in real-world Java programs are optimized successfully.

```
1  int sumOfSquaresEven(int[] v) {
2    int result = 0;
3    for (int i = 0; i < v.length; i++)
4      if (v[i] % 2 == 0)
5        result += v[i] * v[i];
6    return result;
7  }
```

(a) Imperative style.

```
8  int sumOfSquaresEven(int[] v) {
9    return IntStream.of(v)
10           .filter(x -> x % 2 == 0)
11           .map(x -> x * x)
12           .sum();
13 }
```

(b) Functional style, using a stream pipeline.

Figure 4.1: Two variants of computing sums of even squares.

4.1 Introduction

Functional programming is no longer a niche programming paradigm. Although classic functional languages may remain mainly of academic interest only, functional language features are being integrated into mainstream languages, most importantly Java. Version 8 of Java was released in 2014 and included features such as lambda functions and the Stream API [61, 62], which enables functional-style processing of data. A 2017 study found that the adoption of lambda expressions is growing, and that they are mostly used for behavior parameterization such as in stream pipelines [50]. As a simple example, fig. 4.1 shows two ways of computing sums of even squares: (a) using traditional imperative-style iteration and mutable state, and (b) using a functional-style stream pipeline. A stream pipeline consists of a source, in this case an array of integers, operations to be performed on the elements of the stream, here `filter` and `map`, and a terminal operation, such as `sum`. The advantages of functional programming are well known; most importantly, once familiar with this paradigm, the declarative style and absence of side-effects tend to make code easier to read and write than the imperative alternatives, especially for more complex computations.

Despite this advance in language and library design, programmers sometimes avoid using streams for performance reasons. The authors of the 2017 study interviewed a developer from the Open Source project Cassandra on adoption of lambda expressions, who mentioned that *"...Unfortunately, we quickly realized that Streams and Lambdas were pretty bad from a performance point of view. Due to this fact, we stopped using them in hot path."* A developer at Oracle working on the HotSpot Java compiler wrote: *"In order to get the full benefit from JDK 8 streams we will need to make them optimize fully"* [70]. In 2014, Biboudis et al. [8] measured the performance of stream APIs in different languages, including Java 8 and Scala, on seven micro-benchmarks that compare stream pipelines with traditional imperative data processing. They found the Java 8 stream implementation to be the most mature with regards to performance, but also that the baseline imperative-style alternatives were much faster. For example, for their benchmark `sumOfSquaresEven`, which performs the computation shown in fig. 4.1, the stream approach suffered from a 60% performance degradation compared to the baseline implementation. For pipelines that include the `flatMap` operation, the performance overheads were even larger, and a later study shows performance losses that grow quickly in the number of intermediate pipeline operators [39].

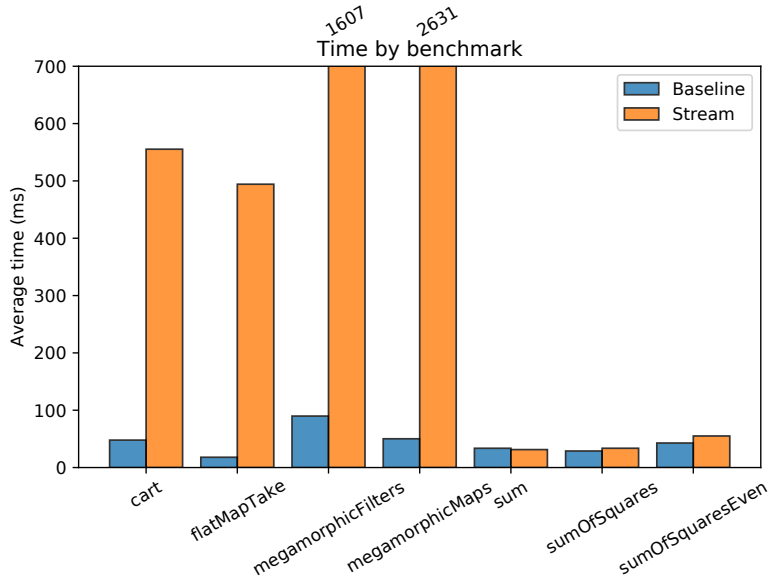


Figure 4.2: Performance of baseline imperative implementation versus sequential Java streams.

Interestingly, today – six years after the experiments by Biboudis et al. – their conclusions still hold, despite improvements in compilers and virtual machine technology. We have replicated their study in Java 13 using the OpenJDK Server VM (build 13+33) with default settings on a machine with an Intel i7-8700 @ 4.6GHz processor and 16 GB of memory. The results are shown in fig. 4.2.[1] As an extreme case, the `megamorphicMaps` benchmark is still around 52× slower when using streams compared to the imperative-style baseline.

One of the strengths of stream pipelines is that it is often easy to switch to parallel processing and thereby exploit modern multi-core CPUs. Although this may reduce the computation time, it is not an ideal solution. It wastes cores, and even for trivially parallelized pipelines, there is typically still a substantial overhead [8]. Moreover, parallel processing does not work well with stateful stream operations, such as `sorted`, or computations with side-effects.

The problem with abstraction overhead of stream processing is well known also for other programming languages than Java. This has motivated the development of, for example, the *strymonas* library for Scala and OCaml [39], LinqOptimizer for C# and F# [65], and ScalaBlitz for Scala [67], however, those approaches are based on meta-programming capabilities that are not available in Java.

Since the early versions of Java, the main approach to code optimization has been as part of just-in-time (JIT) compilation in the virtual machine. Few ahead-of-time (AOT) optimizations are performed by `javac`, since it is believed that the JIT

[1]The micro-benchmarks from Biboudis et al. [8] can be found at `https://github.com/strymonas/java8-benchmarks/blob/master/src/main/java/benchmarks/S.java`.

optimizer is able to make smarter decisions at run-time based on profiling information. In principle, at run-time the optimizer could be able to deduce that a stream pipeline can be transformed into a for-loop to yield optimal performance, however, the experiments mentioned above show that this is often not the case in practice, even for simple stream pipelines. By manually tuning the HotSpot JIT settings for the `megamorphicMaps` benchmark to inline much more aggressively, we find that a $2\times$ speedup can be obtained, but it is still an order of magnitude slower than the baseline. Also, substantial modifications to the JIT settings compared to the defaults can of course degrade performance for other code. Although promising results have been obtained for the Graal JIT compiler on Scala code [68], AOT optimization techniques have advantages compared to JIT optimizations. First, a JIT compiler has to make fast decisions about what and when to optimize while running the program, whereas an AOT optimizer can be given time to perform more precise whole-program analysis. Second, JIT optimizations are known to be unpredictable, while AOT techniques allow the developer to know before program execution whether an optimization attempt succeeds. Third, new AOT optimizations can be deployed, for example in mobile apps, without requiring modifications to the JVM installations. These observations suggest that it may be time to start pursuing AOT optimization techniques for Java, to reach the full potential of functional-style Java code, most importantly for stream pipelines.

By the use of bytecode-to-bytecode transformations driven by a static program analysis, we combine the best of two styles of programming: the conciseness of functional-style stream pipelines at source-code level, and the efficiency of low-level imperative code at run-time. Among the salient features of our approach are that it does not require adding new Java language features or modifications of the application source code, it does not depend on API-specific knowledge (we demonstrate that it works on both push- and pull-style stream APIs without any adaptation), and it is predictable in the sense that the programmer can be informed ahead-of-time whether optimization succeeds for a given stream pipeline. Furthermore, as the transformations and the static analysis work on Java bytecode, this optimization technique is easy to integrate into existing program development processes.

In summary, the contributions of this paper are:

- We propose an ahead-of-time Java bytecode optimization technique that targets stream pipelines in Java code to make them as efficient as hand-written imperative code. Specifically, we demonstrate that applying a combination of well-known program transformations suffices to reach this goal, most importantly, method inlining and stack allocation.

- We present a static program analysis that simultaneously performs type and pointer analysis for driving the program transformations.

- We report from an experimental evaluation showing that applying the optimization to a suite of 11 micro-benchmarks makes 10 of them as fast as hand-written imperative code, in several cases leading to more than $10\times$ speedup, and that

77% of 6 879 stream pipelines found in real-world Java programs are optimized successfully. The evaluation also demonstrates that the approach is not limited to Java's push-style streams but also works for a pull-style stream API, although with potential for improvements of the static analysis.

4.2 Background: Pull- and Push-Style Stream APIs

Stream APIs can be implemented in two different styles. A *pull-style* stream API follows the iterator protocol, with a method `hasNext` for querying whether the stream has more elements and a method `next` for pulling out the next element from the stream. The iteration through the elements of the stream is controlled by the terminal operation, and each operation in the pipeline thus pulls the elements one at a time from its predecessor.

In a simple pull-style stream API, the `map` intermediate operation, which applies a given function to each element of the stream, can be implemented in Java as shown in fig. 4.3. The function allocates a new `PullStream` object to represent the intermediate mapping operation, which becomes the new head of the pipeline. When elements are queried from this head, it extracts an element from its predecessor in the pipeline (using `PullStream.this.next` to refer to the method of the outer class) and applies the supplied function to it before it is returned.

A *push-style* stream API instead includes a single method that takes a consumer action to apply to each element in the stream. When executing the stream pipeline, the source operation controls the iteration by pushing every element in the underlying data source to its consumer action until the data source is empty or until the pipeline terminates early (for example, the `findFirst` operation usually does not have to look at all the elements).

In a push-style stream API, the `filter` intermediate operation, which filters out elements that do not satisfy a given predicate, can be implemented as shown in fig. 4.4. A new `PushStream` object is allocated to form the new head of the pipeline.

```
14 public abstract class PullStream<V> implements Stream<V> { ...
15     public <U> Stream<U> map(Function<? super V, ? extends U> f)
           {
16         return new PullStream<U>() {
17             protected U next() {
18                 return f.apply(PullStream.this.next());
19             }
20             protected boolean hasNext() {
21                 return PullStream.this.hasNext();
22             }
23         };
24     } ...
25 }
```

Figure 4.3: The `map` intermediate operation in a pull-style stream API.

```
26 public abstract class PushStream<V> implements Stream<V> { ...
27     public PushStream<V> filter(Predicate<? super V> p) {
28         return new PushStream<V>() {
29             protected void exec(Consumer<? super V> c) {
30                 PushStream.this.exec(x -> {
31                     if (p.test(x)) c.accept(x);
32                 });
33             }
34         };
35     } ...
36 }
```

Figure 4.4: The `filter` intermediate operation in a simple push-style stream library.

When we execute this pipeline by calling `exec` on the final stream object, the filtering operation constructs a new consumer and passes it to its predecessor in the pipeline (using `PushStream.this.exec` to refer to the method of the outer class). When this consumer is invoked with an element, it only forwards it to the next consumer if the element satisfies the predicate that was given to the `filter` function.

The stream API in Java's standard library is push-style, whereas Scala's views and C#'s Language-Integrated Queries (LINQ) are pull-style [8].

The Java stream implementation is quite complex. It provides specialized stream pipelines for the `int`, `long` and `double` Java primitives to avoid boxing at run-time, and pipelines can be executed in parallel using multiple threads. Additionally, certain characteristics of pipelines are recorded to perform further optimizations at run-time. For example, in the pipeline `list.stream().sorted().sorted().collect(Collectors.toList())`, the second sorting operation is skipped, since the library infers that the stream is already sorted after the first one.

One of the main reasons why stream pipelines are less efficient than their imperative-style counterparts is that executing a stream pipeline involves many virtual calls. A Java stream pipeline with N elements and depth K will accumulate up to $N \times K$ virtual calls just to push the elements through the pipeline [39]. Our optimization technique builds on the key observation that fully inlining the consumer chain will reduce this to a constant number of calls depending on the stream source. Method inlining (also called inline expansion) is a classic compiler optimization technique that replaces a call with the body of the method being called, with parameters and return value flow properly substituted to preserve the program semantics [4, 24].

If we take a deeper look into Java's stream implementation we see that its streams are backed by *spliterators*, which are similar to iterators but support more advanced features, such as splitting a data source into smaller chunks for parallel computation. Every Java class that implements the `Collection` interface inherits a default implementation of a spliterator backed by the collection's iterator implementation. Stream pipelines eventually end up calling the `forEachRemaining` method on the backing spliterator (unless they stop early due to short-circuiting operations such as `findFirst`) to push every element in the collection through a provided consumer

```
40  public class ArrayList<T> extends ... implements ... {
41    transient Object[] elementData;
42
43    @Override
44    public Spliterator<T> spliterator() {
45      return new ArrayListSpliterator(...);
46    }
47
48    final class ArrayListSpliterator<T>
49     implements Spliterator<T> {
50      public void forEachRemaining(Consumer<? super T> action) {
51        int i, hi, mc;
52        Object[] a = elementData;
53        ...
54      }
55      ...
56    }
57    ...
58  }
```

Figure 4.5: Excerpt of the spliterator implementation in Java's `ArrayList` class.

function. For the default iterator-backed spliterator, this translates roughly into this code:

```
37  while (it.hasNext()) consumer.accept(it.next());
```

This is expensive, as it requires three virtual calls per element, even for a pipeline with zero intermediate operations. Therefore, to achieve better performance, collections can provide their own spliterator implementation. For example, the `ArrayList` class provides an efficient `forEachRemaining` spliterator implementation that is essentially a while loop over the internal array:

```
38  while (index < elementData.length)
39      consumer.accept(elementData[index++]);
```

This means that if we can fully inline the execution of such a stream pipeline, we expose a primitive while loop with a single virtual call per element. If we can furthermore inline calls to the `consumer.accept` method, the code we are left with will resemble a hand-written imperative loop construct.

Inlining a method call is a relatively simple program transformation in itself. The key to be able to inline the relevant calls ahead-of-time is precise type information to enable virtual call resolution. The static analysis we present in section 4.5 is designed to provide this information.

One complication to inlining is that spliterators sometimes rely on private fields in the source collection, in which case naively inlining will violate Java's access rules. An example is the spliterator in Java's `ArrayList` class, shown in fig. 4.5, where the `forEachRemaining` method accesses the field `elementData`, which is package-private (i.e., Java's default access mode). This prevents inlining the method into another package. A similar situation may occur when the default iterator-backed spliterator is used, since the implementations of `hasNext` and `next` in the source

```
59 public interface IntStream<StreamT extends IntStream> {
60     /* Intermediate operations */
61     StreamT map(IntUnaryOperator f);
62     StreamT filter(IntPredicate p);
63     StreamT flatMap(IntFunction<? extends StreamT> f);
64     StreamT limit(long maxSize);
65     /* Terminal operations */
66     void forEach(IntConsumer c);
67     int reduce(int initial, IntBinaryOperator r);
68 }
```

Figure 4.6: A simple `IntStream` interface, which can be implemented either pull-style or push-style.

collection may also rely on private fields. This is not a concern for any of the most widely used collections in Java's standard library, but it can be an issue for non-standard stream sources. In section 4.6 we discuss different options for how to handle these situations.

To be able to study optimization opportunities in a more controlled environment and to present manageable examples in the following sections, we have created a simple implementation of a push-style stream library with an API that is similar to that of Java's streams and also to stream implementations studied in related work [9]. This library suffers from the same performance issues as Java's standard stream library (see section 4.8). Its API is shown in fig. 4.6. The library additionally supports pull-style streams, which allows us to explore the flexibility of our optimization techniques also for such a fundamentally different kind of stream API than the one in Java's standard library.

Our current focus is on optimizing sequential (i.e., non-parallel) stream pipelines, since those are by far the most common in practice. (In 28 randomly selected open source Java projects from the RepoReapers dataset [57] that we could build we found 6879 sequential stream pipelines and 49 parallel stream pipelines. A recent study by Khatchadourian et al. [37] confirms this finding.) Still, sequential stream pipelines are often used in multi-threaded applications, so we cannot assume a single-threaded execution environment when designing optimization techniques.

Another crucial observation we can exploit when optimizing stream pipelines is that the entire construction and execution of a typical pipeline take place locally within a single method. Objects of type `Stream` rarely appear as arguments or return values at calls to other methods than those in the stream library, nor are such objects stored in data structures on the heap. We have made a quantitative study of the top 100 Java projects on GitHub to experimentally verify this claim, and found that 93% of calls with streams as parameter or return types are to stream sources, intermediate, or terminal operations, and there is only one field access that involves stream objects per 200 stream operations in the code. Furthermore, all the spliterator objects, consumer objects, and other transient objects are only used internally within the stream operations. This means that all the information stored in these objects can

```
69  private static class PushStream$1<V> extends PushStream<V> {
70      private final PushStream<V> previous;
71      private final Predicate<? super V> predicate;
72
73      PushStream$1(PushStream<V> previous, Predicate<? super V>
            predicate) {
74          this.previous = previous;
75          this.predicate = predicate;
76      }
77
78      protected void exec(Consumer<? super V> c) {
79          previous.exec(x -> {
80              if (predicate.test(x)) c.accept(x);
81          });
82      }
83  }
84
85  public PushStream<V> filter(Predicate<? super V> p) {
86      return new PushStream$1<>(this, p);
87  }
```

Figure 4.7: The `filter` intermediate operation from fig. 4.4, flattened such that the inner class is now lifted outside the `filter` method, similar to the structure of the bytecode.

be placed on the stack instead of in the heap, thereby eliminating the object allocations and reducing the need for garbage collection. As for inlining, this optimization, called stack allocation, is widely used and well understood [17, 66].

The Java JIT compiler already tries to perform stack allocation,[2] but the escape analysis used in the Java JIT to drive stack allocation is limited by being intraprocedural only. We exploit the fact that stack allocation works even better in the AOT setting that allows more precise analysis, together with the aggressive inlining strategy described above. Conversely, stack allocation can also boost inlining. For example, if a method accesses private fields, it cannot be inlined at call sites in other classes, however, if the fields are moved to local variables then inlining can be performed without violating Java's access control mechanisms.

To understand how stack allocation can apply to stream pipelines, consider the `filter` operation from fig. 4.4. It contains an inner class, which the Java compiler lifts outside the method. The free variables of the inner class, `PushStream.this` and `p`, then become fields that are set by the constructor. Java code that roughly corresponds to the resulting bytecode is shown in fig. 4.7. Here, it is clear that every access to `PushStream.this` and `p` in the original method actually involves fields in objects in the heap. This heap allocation cannot be converted into stack allocation without information about the code that calls the `filter` method and uses of the resulting stream object via its `exec` method. By statically analyzing the entire pipeline,

[2] `https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#escapeAnalysis`

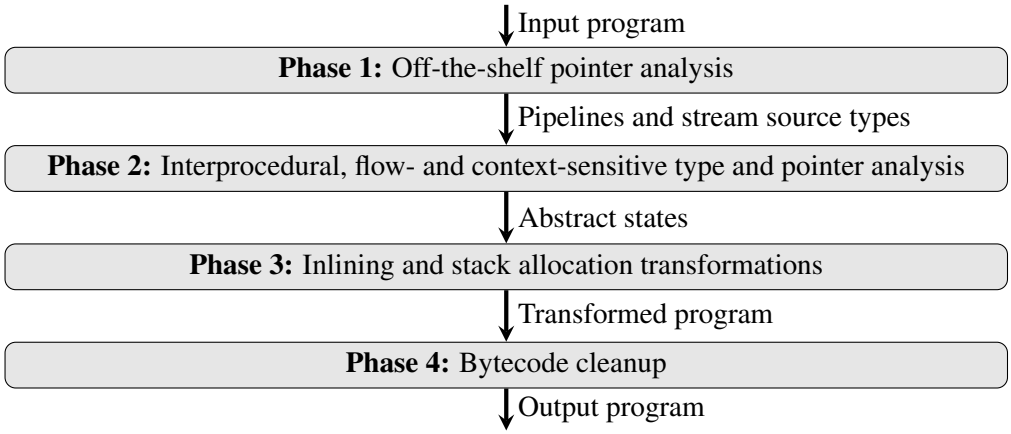


Figure 4.8: Flow diagram of the approach.

our approach can obtain the required information. Also note that the `exec` method in fig. 4.7 cannot be inlined unless we also perform stack allocation of `PushStream$1`, because the fields are declared as `private`.

In summary, these observations suggest that method inlining and stack allocation, which are two classic optimization techniques, can be effective together in AOT optimization of stream pipelines.

4.3 Approach Overview

Figure 4.8 shows the structure of our approach. We start the optimization process by analyzing the compiled program with an off-the-shelf pointer analysis [11, 45, 79]. The purpose of this phase is to find the segments of the bytecode that correspond to stream pipelines in the program and to find the concrete types of the stream sources.

We then process each stream pipeline individually. As mentioned in section 4.2, stream pipelines rarely span multiple methods, and the stream objects are rarely stored in the heap, so tracking the flow of stream objects is trivial (we skip pipelines where this is not the case). For each pipeline we perform a flow- and context-sensitive type and pointer analysis. The result of this analysis consists of an abstract state for every analyzed control flow graph node (in JVM bytecode, a node corresponds to a bytecode instruction), for each call context. Informally, an abstract state maps each local variable and object field to an abstract value, which is a pair of a Java type and an abstract points-to value. For the types, we distinguish between *abstract types* τ and *concrete types* $\overline{\tau}$ [2], where an abstract value with type τ can represent any object that is a subtype of τ, while $\overline{\tau}$ only represents objects that have exactly the type τ. The abstract points-to values use allocation-site abstraction [16] as the heap model. The analysis is explained in more detail in section 4.5.

We then use facts from the analysis result in the next phase to guide the optimization transformations. The type information allows us to resolve calls for the inlining transformation, while the pointer information is used in the stack allocation transfor-

mation to redirect field accesses on stack allocated objects to the corresponding local variables. After the pipeline has been transformed, we feed it to a cleanup phase that can remedy some inefficiencies that are inherent to the two transformations.

It may be the case that the analysis aborts. This happens when it can determine that the analysis result would not allow optimizations to take place. If, for example, the analysis discovers that a pipeline object flows out of the boundary of the analyzed method, for instance to a static field, then after this point the analysis (and therefore also the transformation) cannot make any useful assumptions about the state of the object. Another condition for aborting is over-approximation of an unanalyzable call that leads to an unusable analysis result. These and other cases are described in more detail in section 4.5.

Example To get an intuitive understanding of the approach before we explain the details in the following sections, we can apply it to the example stream pipeline shown in fig. 4.9a. It is presented as Java source code for readability although the actual technique works on Java bytecode.

The pre-analysis finds a stream pipeline in the `sum` method. The initial abstract state for the main analysis is seeded with abstract values from the pre-analysis for the local variables and the stack. The method contains one local variable for the argument `v`, which is given the abstract value $(\texttt{int}[], \mathsf{unrelated})$. Here, unrelated denotes a pointer value that cannot refer to any allocation site that appears during the analysis and whose field values are not tracked in the abstract heap. Since the stream source is constructed by a static method call in this example, it does not require additional type information to be resolved. (We show an example in section 4.4 that needs the pre-analysis to infer the concrete type of the stream source object.)

The first instructions encountered during the main analysis phase correspond to the call to `IntPushStream.of(v)`. We resolve this call by finding the static method on `IntPushStream` with the correct signature and continue by analyzing the resolved method:

```
88  public static IntPushStream of(int[] arr) {
89      return new IntPushStream() {
90          public void exec(IterConsumer c) {
91              int i = 0;
92              while (i < arr.length) c.accept(arr[i++]);
93          }
94      };
95  }
```

This `of` method constructs an instance of an anonymous subclass of `IntPushStream`. Following Java conventions this subclass could be named `IntPushStream$0`. We update our abstract state with this allocation site named ℓ_1 and continue by analyzing the constructor of `IntPushStream$0` (not shown here) and see that the implicitly passed `arr` parameter is stored as a field that is also named `arr` in the subclass. The abstract value that is returned from `IntPushStream.of` is $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$, and the heap part of the new abstract state maps ℓ_1 to $\{\texttt{arr} \mapsto (\texttt{int}[], \mathsf{unrelated})\}$. The

```
96  public static int sum(int[] v) {
97      IntPushStream stream = IntPushStream.of(v);
98      return stream.reduce(0, Integer::sum);
99  }
```

(a) An example method, sum, containing a stream pipeline.

```
100 public static int sum(int[] v) {
101     // inlined IntPushStream.of
102     int[] IntPushStream_of_arr = v;
103     IntPushStream stream = null;
104     int[] IntPushStream$0_arr = IntPushStream_of_arr;
105     // inlined IntPushStream.reduce
106     IntPushStream IntPushStream_reduce_this = stream;
107     int IntPushStream_reduce_initial = 0;
108     IntBinaryOperator IntPushStream_reduce_r = null;
109     Reducer IntPushStream_reducer = null;
110     int Reducer_state = IntPushStream_reduce_initial;
111     IntBinaryOperator Reducer_operator = IntPushStream_reduce_r
            ;
112     // inlined IntPushStream$0.exec
113     IntPushStream$0 IntPushStream$0_exec_this = (
            IntPushStream$0) IntPushStream_reduce_this;
114     IterConsumer IntPushStream$0_exec_c = IntPushStream_reducer
            ;
115     int IntPushStream$0_exec_i = 0;
116     while (IntPushStream$0_exec_i < IntPushStream$0_arr.length)
             {
117         // inlined Reducer.accept
118         Reducer Reducer_accept_this = (Reducer)
                IntPushStream$0_exec_c;
119         int Reducer_accept_v = IntPushStream$0_arr[
                IntPushStream$0_exec_i++];
120         // inlined Integer.sum
121         int Integer_sum_a = Reducer_state, Integer_sum_b =
                Reducer_accept_v;
122         Reducer_state = Integer_sum_a + Integer_sum_b;
123     }
124     return Reducer_state;
125 }
```

(b) The sum method after inlining and stack allocation.

```
126 public static int sum(int[] v) {
127     int state = 0, i = 0;
128     while (i < v.length) state += v[i++];
129     return state;
130 }
```

(c) The resulting sum method after the cleanup phase.

Figure 4.9: Optimization of a stream pipeline.

next instructions in `sum` allocate the lambda argument to the `reduce` function, causing the abstract state to be updated with a new allocation site, ℓ_2. The abstract value of the receiver of the `reduce` call is $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$ and contains precise type information such that we can resolve the call according to Java's virtual method invocation semantics.[3] We thus continue by analyzing `IntPushStream.reduce`, shown below, where the abstract values of the receiver and arguments are $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$, $(\overline{\texttt{int}}, \top)$, and $(\overline{\lambda_0}, \ell_2)$, respectively, where λ_0 denotes the type of the object created for the method reference `Integer::sum`.

```
131  public int reduce(int initial, IntBinaryOperator r) {
132      Reducer reducer = new Reducer(initial, r);
133      this.exec(reducer);
134      return reducer.state;
135  }
```

Continuing analysis in this method, after allocating the `Reducer` at allocation site ℓ_3, we have the abstract heap

$$[\ell_1 \mapsto \{\texttt{arr} \mapsto (\texttt{int}[], \mathsf{unrelated})\}, \ell_2 \mapsto \{\},$$
$$\ell_3 \mapsto \{\texttt{state} \mapsto (\overline{\texttt{int}}, \top), \texttt{reducer} \mapsto (\overline{\lambda_0}, \ell_2)\}]$$

and the abstract values of `this`, `initial`, and `r` are $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$, $(\overline{\texttt{int}}, \top)$, and $(\overline{\lambda_0}, \ell_2)$, respectively. The `Reducer` carries the `state` field to keep track of the running sum while the pipeline executes, and a reference to the `reducer` method. The purpose of that method is to compute a new state from a stream element and an old state, as a left-fold operation.

To resolve the call to `exec`, the analysis looks up the abstract value of the receiver (`this`), which is $(\overline{\texttt{IntPushStream\$0}}, \ell_1)$ in this case, so it can continue the analysis in `IntPushStream$0.exec` where the abstract value of the first argument is $(\overline{\texttt{Reducer}}, \ell_3)$. Inside `exec` (see lines 90–93) the field `arr` of ℓ_1 is accessed twice. The analysis can precisely resolve these accesses using the abstract state since the abstract points-to value of `this` is ℓ_1. The call to `Reducer.accept` is resolved and the analysis continues in that method:

```
136  public void accept(int v) {
137      state = reducer.applyAsInt(state, v);
138  }
```

Here it is even more crucial that the analysis has precise type and points-to information for `this`, as this allows it to look up the value of `reducer` on ℓ_3 in the abstract state and resolve the call to λ_0`.applyAsInt`. After finishing the analysis, we have the information necessary to unambiguously resolve the calls at every analyzed call site.

The following phases perform optimizing transformations on the analyzed pipeline. Like the analysis it operates on stack-based Java bytecode, but we will outline the transformation as if it happens directly on Java source code. The first transformation

[3] https://docs.oracle.com/javase/specs/jvms/se13/html/jvms-6.html#jvms-6.5.invokevirtual

phase applies inlining and stack allocation transformations. Local variables are used in place of the object's fields and are also allocated for the parameters of inlined methods.

We can look into how the analysis result is used to transform the call to `IntPushStream.of(v)` in fig. 4.9a. The analysis resolved the callee such that it can be inlined into the `sum` method:

```
139 int[] IntPushStream_of_arr = v;
140 IntPushStream stream = new IntPushStream() {
141     public void exec(IterConsumer c) {
142         int i = 0;
143         while (i < IntPushStream_of_arr.length)
144             c.accept(IntPushStream_of_arr[i++]);
145     }
146 };
```

Notice that a fresh local variable has been generated for the `arr` parameter.

The next step is to allocate `IntPushStream$0` on the stack instead of on the heap (see line 140). First, the **`new`** instruction is replaced with **`null`** to preserve the operand stack layout. The class has one field of type `int[]` so a local variable is allocated for it (`IntPushStream$0_arr` at line 149 below). The variable is associated with the allocation site ℓ_1 of the `IntPushStream$0` object, and the constructor is inlined. Inside the constructor of `IntPushStream$0`, the array is stored as a field on the object. According to the abstract state for the field write instruction, the object that is written to is the object allocated at site ℓ_1. The field write instruction can therefore be redirected to write to the local variable allocated for the object (see line 150):

```
147 int[] IntPushStream_of_arr = v;
148 IntPushStream stream = null;
149 int[] IntPushStream$0_arr = null;
150 IntPushStream$0_arr = IntPushStream_of_arr;
```

Whenever the transformation later finds an instruction that accesses the `arr` field of ℓ_1, it is similarly replaced with an instruction that instead accesses `IntPushStream$0_arr`.

The result of the transformation phase is shown in fig. 4.9b. This transformed method can be executed as is but is rather large and filled with redundancies. The last transformation phase aims to reduce the code size and the number of local variables in the resulting code. It does so by identifying and removing duplicate aliasing local variables, unused variables, and redundant bytecode instructions. After these transformations, we end up with the code shown in fig. 4.9c. Notice that the resulting code is a simple while-loop that iterates over the array, without any virtual calls, similar to what a programmer would likely write if not having streams available.

In the following sections, we describe how each of the four phases work more generally, and with more details about the analysis and transformations.

4.4 Phase 1: Pre-Analysis

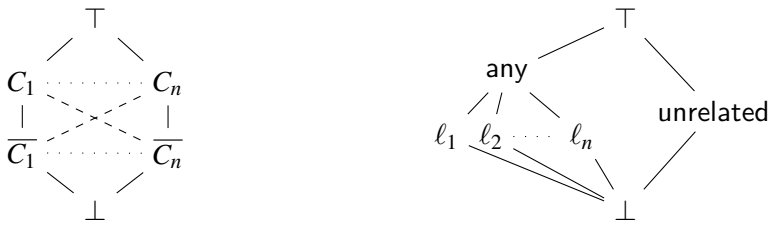
The optimization process begins with a preliminary analysis. Its goals are (1) to identify stream pipelines within the analyzed program, and (2) to restrict the set of possible concrete types for stream sources. This information allows us to subsequently use an expensive analysis, which is specialized for guiding our optimizations, only at the program points where it is needed. The concrete types for the stream sources are used for seeding the analysis in phase 2.

Consider the example stream pipeline marked with gray in fig. 4.10. By simply using the type information available in the Java bytecode of the compiled program, it is trivial to find all local variables of type `Stream`, which allows us to recognize the segment of bytecode constituting the pipeline. (In case the stream objects are passed as parameters or return values of non-application methods, or they are stored in or retrieved from fields in objects, we simply give up optimizing the pipeline, as mentioned earlier; this can be improved in future work.) Application code, such as the two lambdas in fig. 4.10, is considered outside the pipeline by the main analysis although it is being inlined in the transformation phase. However, we do analyze the (implicit) constructors for the lambdas, to be able to detect if stream objects escape from the pipeline code. An exception is made for the `flatMap` operator where the callback creates a stream object directly involved in the execution of the pipeline, and must therefore be included in the main analysis.

In many cases, the stream source type is trivial to infer (as in the `sum` example in fig. 4.9a), but other cases require information about dataflow. When compiled to bytecode, the call to `list.stream()` in fig. 4.10 simply contains `List` as the receiver of the call. At run-time it is up to the JVM to dispatch the call to the implementation of the concrete type of the receiver of the call, in this case `ArrayList`. It does not require an advanced analysis to figure out that the concrete type of the receiver is indeed `ArrayList` in this simple example, but the receiver is not always allocated in the same method as the stream pipeline, as it could be passed as an argument to the method or reside in a field of an object. To handle such situations, we can apply an

```
151 class Application {
152   private void method(...) {
153     List<Integer> list = new ArrayList<Integer>();
154     // ...
155     boolean anyMatching = list.stream()
156                           .map( x -> x * y )
157                           .anyMatch( x -> x > z );
158     // ...
159   }
160 }
```

Figure 4.10: A stream pipeline (marked) as part of a bigger program. Note that the lambdas are excluded from the code considered by the main analysis.



(a) *Type* lattice for all classes, interfaces, primitive types, and array types $C_1 \ldots C_n$.

(b) *Pointer* lattice for allocation sites $\ell_1, \ell_2, \ldots, \ell_n$.

Figure 4.11: Lattices for type and pointer components of abstract values.

off-the-shelf pointer analysis to statically find the concrete type of the stream source object. Several such tools are available, including Soot [45, 79], Doop [11], and WALA [28]. Instead of simply selecting every non-abstract subclass of the declared type `List` of the `list` variable, such analyzers can narrow the set of possible concrete types by safely over-approximating the dataflow in the program. Usually, this gives us a single concrete type for each pipeline source. In case multiple possible concrete types are found, one possibility is to optimize the pipeline separately for each of them and then branch at run-time based on the actual type. Since we typically only need the type information for a small number of expressions in the program, a demand-driven analysis [77, 79], which only analyzes the relevant part of the code, is a good fit.

4.5 Phase 2: Interprocedural analysis

We express the main analysis as a monotone framework [35], which requires a lattice of abstract states. Abstract values are defined as elements of the lattice

$$\mathit{Values} = \mathit{Type} \times \mathit{Pointer}$$

and *Type* and *Pointer* are illustrated in fig. 4.11. The *Type* lattice contains all concrete and abstract types, for modelling classes, interfaces, primitives, and arrays. The dashed edges between the types represent the class and interface inheritance relations. The least upper bound of two Java types is not always a single unique class or interface, which is why this simple lattice is chosen. In the *Pointer* lattice, any represents a value that can point to any allocation site that occurs in the pipeline. The lattice element unrelated represents values that can point to objects unrelated to the execution of stream pipelines or objects allocated before the analysis entry point. Values that are references to objects are modeled by object labels, $\ell_1, \ell_2, \ell_3, \ldots, \ell_n \in \mathit{ObjectLabels}$. Objects are abstracted by their allocation sites, $\mathit{ObjectLabels} = \mathit{Contexts} \times \mathit{Nodes}$. Here, *Nodes* is the set of control flow graph nodes (i.e., bytecode instructions), and *Contexts* is the set of all call contexts (explained below), so the allocation sites are qualified by contexts (also called heap cloning or context sensitive heap) [60, 76].

An abstract state defines an abstract value for each local variable and operand stack cell (together referred to as the set *Cells*), and it carries an abstract heap that

maps each object label to a map from fields to abstract values:

$$States = \overbrace{(Cells \to Values)}^{\text{Stack and locals}} \times \overbrace{(ObjectLabels \to Fields \to Values)}^{\text{Heap}}$$

To achieve context sensitivity we apply the well known call-string technique with unbounded length [74]. This allows us to precisely analyze stream pipelines of arbitrary depth. Such an extreme choice of context sensitivity is of course not scalable to Java code in general, but stream pipelines are relatively small. For modeling object constructions, we pick the entire current call context as context for the object labels. The analysis is also flow sensitive, so the full analysis lattice has an abstract state for each context and control flow graph node:

$$Contexts \to Nodes \to States$$

The analysis is invoked for each pipeline found in phase 1. Such a pipeline is marked by a range of bytecode instructions that contain all instructions relevant to its execution. Thus the analysis can start at the first of these instructions when initialized with a sensible initial abstract state. This state contains the concrete type information from the pre-analysis necessary to resolve the call to the stream source. The analysis then proceeds to analyze the pipeline code. If a critical loss of precision occurs on the way, the analysis aborts and no optimization of the pipeline is performed.

The transfer functions of the intraprocedural parts of the analysis are straightforward. Generally they model the modification to the abstract operand stack and local variables after executing bytecode instructions. For instance, the `getfield` instruction on a field F of type τ looks at the topmost value (τ', p) of the abstract stack and proceeds by case analysis to figure out which value v to replace it with:

$$v = \begin{cases} lookup(p)(F) & \text{if } p \in ObjectLabels \\ \bigsqcup_{\ell \sqsubseteq p \,\wedge\, filter(\ell,\tau')} lookup(\ell)(F) & \text{if } p = \mathsf{any} \\ (\tau, p) & \text{if } p \in \{\top, \mathsf{unrelated}\} \\ (\bot, \bot) & \text{if } p = \bot \end{cases}$$

The function $lookup(\ell)(F)$ looks up the abstract value of field F in abstract object ℓ in the current abstract state. In the first case, $p \in ObjectLabels$, we simply look up the field value, which precisely captures the semantics of `getfield`. If $p = \mathsf{any}$ then the result value is the least upper bound of the abstract values for that field on all relevant objects. The predicate $filter(\ell, \tau')$ filters the set of object labels ℓ according to the type τ': If τ' is a concrete type then only abstract objects of exactly that type are included, otherwise abstract objects that are subclasses of τ' are included. If $p \in \{\top, \mathsf{unrelated}\}$, then v cannot be refined further than (τ, p), as p could point to an object that the analysis does not track.

For `putfield` instructions, the two topmost values on the stack are popped. Let (τ_o, p_o) and (τ_v, p_v) denote the abstract values of the object reference and assigned value, respectively. If $\mathsf{unrelated} \sqsubseteq p_o$ and there exists an object label ℓ such that

$\ell \sqsubseteq p_v$ the analysis aborts, as ℓ can escape from the analyzed part of the program. If $p_o \in \mathit{ObjectLabels}$, a strong update on the object can be performed,[4] otherwise a weak update on all the object labels $\ell \sqsubseteq p_o$ that have the corresponding field is performed.

For the interprocedural part of the analysis we define transfer functions for method calls. The first part of a method call is to resolve the callee. If the call instruction is `invokestatic` or `invokespecial` this is easy,[5] otherwise the callee depends on the run-time type of the receiver. If the type component of the abstract value of the receiver is precise, then we can use Java's virtual method lookup procedure. If not, we might be able to exploit that the targeted method is *final* or that the declaring class is *final*. Otherwise the analysis resorts to over-approximation without involving the callee body, as follows.[6] If any value (τ, p) flows into an over-approximated call (either as receiver or argument) where there exists some $\ell \in \mathit{ObjectLabels}$ such that $\ell \sqsubseteq p$ and $\mathit{filter}(\ell, \tau)$ holds, then the analysis aborts, as the method could modify the full reachable heap from this object, leading to a very imprecise heap. Otherwise, if the callee can be uniquely resolved, analysis continues in the resolved method in a new context with the current abstract heap. After the analysis of the method finishes, we merge the abstract states at all reachable return instructions in the method and continue analysis in the caller with the merged abstract heap. The return value is the topmost value on the stack in the merged state.

It is possible to construct stream pipelines whose structure is not statically fixed, for example by applying a stream operation conditionally as in the following example.

```
161  IntStream s = /* some source */;
162  if (shouldSquare) s = s.map(x -> x * x);
163  return s.filter(x -> x % 2 == 0).sum();
```

This causes the analysis to abort due to a failed resolution of a call target, in this case at a call that appears as part of the terminal operation, no matter if Java's stream library or the simple push- or pull-style implementation described in section 4.2 is used.

Callbacks from the stream library to the application code, such as the lambdas in figs. 4.1b, 4.9a and 4.10, are not analyzed unless it is necessary. These must be analyzed if a pipeline object flows into such a method (the lambda can capture a reference to the pipeline), or if it is the callback from a `flatMap` operator.

Calls to methods implemented in native code are handled by over-approximating as explained above. To prevent this from aborting the analysis in common cases we use custom models for a few core methods in the Java standard library (e.g., `Class.getName` and `System.arraycopy`). Other typical obstacles to sound and precise static analysis for Java, such as reflection or dynamic class loading, are not a concern, because we only apply the analysis to the stream library code, and stream

[4] Strong updating [16] is sound in this situation because of the use of flow sensitivity and full context sensitivity.

[5] `invokestatic` is a direct method call while `invokespecial` resolves the callee by traversing the superclasses of the enclosing class until a matching method is found.

[6] We could instead apply some variant of Class-Hierarchy Analysis [23] to find potential callees and merge the results of analyzing those methods.

libraries do not use such mechanisms. (If the analysis should encounter use of such mechanisms, it simply aborts.)

Context-sensitive analysis with unbounded call strings may diverge for programs that contain recursion. It is not trivial to detect ill-natured recursion, as the call stack may legitimately contain the same method multiple times during the execution of a stream pipeline, if the pipeline contains multiple instances of the same intermediate operation. To ensure termination, we therefore abort the analysis if the length of a call string exceeds 1 000. This bound is well above what is needed to admit analysis of most pipelines.

With this analysis lattice and these transfer functions, the analysis runs using a standard worklist algorithm that repeatedly applies the transfer functions until either a fixed-point is reached, in which case we proceed to the transformation phase, or the analysis aborts due to one of the conditions described above. In summary, the realistic situations where the analysis aborts are (1) an object created in the analyzed part of the code may escape that part of the code (this happens for around 2% of the pipelines in our experiments, see section 4.8), (2) a call target cannot be resolved with sufficient precision (happens for around 14% of the pipelines), and (3) recursion causing the analysis to diverge (happens for less than 1% of the pipelines).

Handling Java's Stream Library The analysis presented above suffices for simple stream libraries, such as the one described in section 4.2, but not for more complex ones. As mentioned earlier, the Java standard library stream implementation is quite complex and uses different code paths for sequential and parallel computation, and for short-circuiting and non-short-circuiting pipelines. To obtain sufficient analysis precision to fuel optimizations, we need to avoid analyzing certain paths that are not taken in actual runs of the code. We achieve this by including constant propagation [15] in the abstract values and by making the analysis control sensitive (also called branch sensitive) to take branch conditions into account for refining abstract values and for eliding dead code.

A common source of precision loss is the use of *stream flags* in the library code. Every stream pipeline has a set of flags that are queried at different stages of execution. If we cannot analyze these queries precisely, the analysis loses too much precision to be useful. The flags are bitmasks that are computed at run-time by the static initializer of the `StreamOpFlag` enum class. The code in a static initializer of the class is run the first time the class is accessed and is mainly used to populate static fields. Since we do not necessarily want to limit ourselves to a whole-program analysis, we cannot make assumptions about when this initialization happens and in what state the static fields of the class are in at the analysis entry point. However, we observe that if the fields have the `final` modifier, they cannot have been reassigned after initialization.

Analyzing `StreamOpFlag`'s initializer statically requires loop unrolling to be precise enough to be useful. We take a simpler approach: Instead we utilize an on-demand dynamic pre-analysis that takes a snapshot of the reachable heap after initializing static fields and preserves abstract values for fields that are marked final. This allows

us to get precise information on `StreamOpFlag` and `StreamOpFlag$Type` enums needed for control sensitivity.

This small extension of the analysis relies on two assumptions. The first is that final fields of pre-analyzed classes are not modified by the client at run-time. This assumption could be violated by clients that use reflection,[7] or by bytecode that is not emitted by the Java compiler. Even though the Java compiler does not allow multiple writes to `final` fields, it is possible to load classes into the JVM that violate this constraint.[8] We also assume that the values of `static final` fields involved in the stream pipeline do not rely on the run-time environment in which they are initialized.

A final trick necessary to enable useful analysis of the Java stream implementation is a model for the standard library method `java.util.stream.AbstractPipeline.wrapSink`. This method is responsible for traversing the stream pipeline from back to front, chaining together consumers (called `Sink`s in Java stream terminology) along the way. This consumer is what the spliterator will send elements into when the pipeline executes, and analyzing the chaining precisely is therefore critical. This method is implemented with a loop instead of with recursion and thus loop unrolling is necessary to analyze the behavior precisely. We have precise enough information to unroll the loop, but we have not extended the analysis to support this in the proof-of-concept implementation, so instead we replace that method with a model that has the loop manually unrolled.

4.6 Phase 3: Inlining and stack allocation

Both kinds of transformations we apply, inlining and stack allocation, are classic compiler optimizations used for decades [17, 24]. In this section we briefly describe how they work, in particular how they depend on each other, how they use the information from the main analysis phase, and what their limitations are. The transformation starts at the entry point of the pipeline and considers each bytecode instruction one-by-one.

Inlining

At a method call instruction the transformation tries to resolve the callee in the same way as the analysis, using the abstract state for this program point. If the callee cannot be uniquely determined, inlining is not applied for the call. Otherwise the callee is resolved to some method with n arguments and m local variables. At the call instruction the Java operand stack must contain at least n values where the top n values will be consumed by the call. At method entry, the callee expects the parameter values to be placed in the local variables numbered 0 to $n-1$. In the caller method, we allocate m new locals for the inlined method. For each argument in reverse order, a `store` instruction is inserted to the appropriate newly allocated local variable before

[7] However, since Java 9 the JVM can disallow all reflective accesses to JDK internal API's. See Relaxed-strong-encapsulation.

[8] See `https://hg.openjdk.java.net/jdk/jdk12/file/06222165c35f/src/hotspot/share/interpreter/rewriter.cpp#l435`

the call instruction. The callee is then recursively transformed where care is taken to remap variable accesses to the allocated variables in the caller. Return instructions are handled by replacing them with an unconditional jump to a fresh label placed at the end of the inlined method.[9] The list of bytecode instructions in the transformed method is then spliced into the caller in place of the call instruction, and the maximum stack size of the caller is adjusted accordingly.

The transformation is easy to apply, but the ability to apply it in the AOT setting can be hindered by Java's access control mechanisms. If the callee is in a different class and/or package than the caller, the callee might be able to access fields, methods, and classes that the caller cannot, for example if they are declared `private`. In this case, inlining the callee would produce code that does not pass Java's runtime encapsulation checks [12, 13]. We return to this issue at the end of the section.

Since the analysis does not cover the whole program, the inlining transformation is only successful if callees can be inlined all the way into the body of the method containing the stream pipeline. If the analysis starts in method f_1 and analyzes a call to f_2 that further calls f_3, only inlining f_3 into f_2 using the abstract states from the analysis would be unsound, as f_2 could have other callers than f_1 where the abstract states do not match the ones we used for the transformation. This implies that the technique is not directly suitable to optimize parallel stream pipelines. In such pipelines, the work of executing the pipeline is delegated to multiple threads, and can therefore not be inlined into the method containing the stream pipeline.

Stack Allocation

Stack allocation can only be done for objects that do not escape their method [17]; the analysis has already checked that property as explained in section 4.5. To be able to perform the transformation in a way that preserves the program semantics, it is also necessary that sufficiently precise pointer information is available from the preceding phase. For this reason, we use the following concept of *stack allocation eligibility*. When an object is allocated in the stack, and its fields are stored in local variables in the call frame instead of on the heap, all instructions that access the object's fields must be appropriately transformed. If this condition cannot be satisfied for a given object (identified by an object label), then the object is ineligible for stack allocation. Eligibility is determined by examining all field access instructions in the analyzed code. At a field access instruction the abstract state can be queried for the abstract value of the object reference, denoted (τ, p). If $p \notin \mathit{ObjectLabels}$ then all object labels ℓ where $\ell \sqsubseteq p$ and $\mathit{filter}(\ell, \tau)$ holds are made ineligible for stack allocation. For such an ℓ the abstract state is not precise enough to ensure that this field access instruction can be redirected to the corresponding local variable. Notice that stack allocation eligibility relies on inlining – all method calls that the potentially stack

[9] The semantics of a JVM call instruction specify that (at most) one value is placed on the operand stack after execution. While the stack is not required to contain only one value at a return instruction, this is the case for all bytecode generated by the Java compiler. Additional measures can be taken to allow for full return semantics.

allocated object flows into must be inlined to ensure that we can translate load and store instructions and inline virtual calls.

After stack allocation eligibility is determined, the transformation starts. It operates on `new`, `putfield`, and `getfield` instructions. Whenever `new` is encountered, the transformation checks whether the allocation site defined by this bytecode instruction is eligible for stack allocation. If this is the case, local variables are allocated for all of the object's fields, and they are associated with the object label. To keep the operand stack layout valid, the `new` instruction is temporarily replaced with an instruction that loads the `null` constant.

At a `getfield` instruction, the abstract state is queried for the abstract value of the object reference, denoted (τ, p) as above. If $p \in \mathit{ObjectLabels}$ and p is eligible for stack allocation, the instruction is transformed into a read to the local variable that was previously allocated for the field. The transformation handles `putfield` instructions similarly.

As mentioned in section 4.3, stack allocation can enable more inlining optimizations. If a candidate method for inlining contains field accesses that would violate Java's access control, inlining can only take place if the object that is accessed is allocated on the stack, such that its fields can be accessed as local variables instead.

In practice, because of the interdependencies between the two transformations, optimizing a given stream pipeline with our technique is usually "all or nothing" – either inlining succeeds for all the methods involved in the pipeline and all the objects created in the process are stack allocated, or the optimization fails entirely.

Handling Private Fields

In section 4.1 we mentioned how spliterator implementations for Java's standard library contain accesses to package-private fields (see fig. 4.5). These accesses are directed at a `Collection` object that should not be stack allocated, either because the object is not allocated within the analyzed method, or because the scope of the analysis would have to be broadened to not only include the stream pipeline but also parts of the application code relevant to the collection object, to be able to carry out the necessary transformations. The consequence is that inlining the spliterator methods will always be prohibited by the rules described above. Not being able to inline the spliterator methods produces a cascade of other optimizations that cannot take place, due to the interplay between them. When this happens, different courses of action are possible:

Inline as much as possible: We can choose to apply only the optimizations that are possible. This is, however, undesirable, as the bulk of the performance benefits of the optimization comes from fully inlining the call to `spliterator.forEachRemaining` and eliminating chains of virtual calls to push elements through the pipeline. This chain starts in the `forEachRemaining` method, and when this method cannot be inlined it disallows inlining of further calls.

Use reflection: Violating field accesses can be circumvented with the use of Java's Reflection API. The use of reflection can have unfortunate performance drawbacks, and is only possible when the application is run in an appropriate Java security context. Since reflection allows us to expose encapsulated fields that are not part of the object's interface, using reflection only works as long as the internal implementation of the object does not change, which could happen between different Java releases.

Exploit the Java module system: Since Java 9 introduced the Platform Module System,[10] it is possible, with appropriate JVM settings, to inject a class into the same module and package as the collection class, to expose private members of the class in a public interface. This does not suffer the same performance drawback as reflection, but still depends on the internal implementation of the class. In addition the JVM must be run with special settings.

Copy the class: We can make a copy of the collection class that publicly exposes its members. This class must be used in place of the original class throughout the application code. This way the application will work no matter the environment in which it runs, and will not suffer any performance drawbacks.

None of these solutions are ideal, but allow the transformation to optimize stream pipelines with collection sources. For our experiments we chose the last course of action as the lesser evil.

4.7 Phase 4: Cleanup

The above transformations, while general, typically introduce a lot of redundant bytecode instructions. The goal of the cleanup phase is to remove some of these redundancies from the transformed method. The `null` values introduced temporarily during the stack allocation transformation (see section 4.6) are also eliminated in this phase. As input, the phase receives the transformed method from the previous phase. It then applies a few simple intraprocedural analyses and transformations described below. We motivate the cleanup techniques with two examples of method bodies that can be shortened. In the first example, method `f` invokes `twice` in line 166:

[10] `https://www.oracle.com/corporate/features/understanding-java-9-modules.html`

```
164  int f(int i) {
165    ILOAD 0
166    INVOKE int twice(int)
167    ...
168  }

169  int twice(int x) {
170    ILOAD 0
171    ICONST_2
172    IMUL
173    IRETURN
174  }
```

```
175  int f_opt(int i) {
176    ILOAD 0
177    ISTORE 1    (removed)
178    ILOAD 1     (removed)
179    ICONST_2
180    IMUL
181    ...
182  }
```

When `twice` has been inlined into `f` as shown on the right, the argument loaded for `twice` will be stored into a fresh local variable that is immediately reloaded and never reassigned. In this case we can remove the store and load instructions (indicated by 'removed' above). In general, such redundancies occur whenever we inline a method that is called with variables as arguments, and the method never assigns to the local variables for those parameters.

Another type of redundancy is introduced in the stack allocation transformation. Consider the following example on the left. A `State` object is created, then `i` is written to its `value` field, and finally the value is read from the field and returned. In this example, the `State` object can be stack allocated.

```
183  class State { public int value; }
184
185  int m(int i) {
186    NEW State
187    DUP
188    INVOKE void State.<init>()
189    ASTORE 1
190    ALOAD 1
191    ILOAD 0
192    PUTFIELD State.value int
193    ALOAD 1
194    GETFIELD State.value int
195    IRETURN
196  }
```

```
197  int m_opt(int i) {
198    ACONST_NULL (removed)
199    DUP         (removed)
200    POP         (removed)
201    ASTORE 1    (removed)
202    ALOAD 1     (removed)
203    ILOAD 0
204    ISTORE 2    (removed)
205    POP         (removed)
206    ALOAD 1     (removed)
207    POP         (removed)
208    ILOAD 2     (removed)
209    IRETURN
210  }
```

On the right, the method is shown after applying the stack allocation transformation, but before cleanup. The `State` class has one field, so a fresh local variable is allocated for it in the method, in this case it gets the index 2. The `putfield` instruction is replaced with a store to the allocated local variable followed by a `pop`. This `pop` is necessary to preserve the operand stack layout. A similar transformation is applied for the `getfield` instruction. With further intraprocedural simplifications, the body of the method can now be reduced to only a `load` of the argument followed by a `return` instruction, making all the other instructions redundant.

Figure 4.9b shows a transformed stream pipeline with redundancies from both the inlining and stack allocation transformations, in Java source code form.

Both inlining and stack allocation introduce a lot of redundant local variables in the transformed method. To eliminate such redundancy, we incorporate a flow-sensitive must-alias analysis similar to the one used in the Scala compiler.[11] This analysis determines which values are guaranteed to be equal for each program point. For instance it can determine that the local variables 0 and 1 must alias at line 178. With this information we can redirect the load instruction to the first local variable. This in turn makes the instructions in lines 176 and 177 dead, so they can safely be removed. We additionally employ other well-known intraprocedural analyses: strongly live variables analysis, nullness analysis, reachability analysis, and sign analysis, and the optimizations they enable, together with a suite of peephole optimizations.

In our benchmarks, the cleanup optimizations reduce the number of local variables in the transformed stream pipeline by a factor of 10 to 40 and the number of bytecode instructions by a factor 10, and thereby enable further optimizations by the JIT. Of course these numbers vary a lot depending on the pipeline in question.

4.8 Evaluation

Our proof-of-concept implementation of the approach, named STREAMLINER, consists of approximately 8 KLOC Java code, building on ASM[12] for bytecode manipulation and analysis.

We evaluate our approach by answering the following research questions:

RQ1: Is the performance of the optimized code comparable to that of hand-optimized code, when applied to micro-benchmarks and using either push- or pull-style libraries?

RQ2: To what extent is the technique able to optimize stream pipelines in real-world Java applications? In cases where it fails, what are the reasons?

The STREAMLINER implementation and experimental data are available at `https://brics.dk/streamliner/`.

RQ1: Performance Evaluation

To answer the first research question we wish to compare the performance of programs before and after optimization. Our approach mainly targets stream pipelines which usually are components of larger programs. To isolate the performance impact of our optimization we evaluate the approach on a suite of 11 micro-benchmarks that consist only of stream pipelines. This suite builds upon micro-benchmarks from previous work [9] and includes a new benchmark that uses the `allMatch` terminal operation. This operation terminates the execution of the pipeline as soon as an element that

[11] `https://github.com/scala/scala/blob/2.13.x/src/compiler/scala/tools/nsc/backend/jvm/analysis/AliasingAnalyzer.scala`

[12] `https://asm.ow2.io/`

Table 4.1: Java Virtual Machines used in the performance evaluation.

Name	Java Version	Build number
Oracle HotSpot VM	8	1.8.0_241
OpenJDK HotSpot VM	13	13+13
GraalVM CE	11	11.0.6+9-jvmci-20.0
Eclipse OpenJ9	13	0.18.0

does not satisfy the supplied predicate is found (it is a short-circuiting operation), and therefore follows an alternative code path in the Java stream library.

We do not include real-world Java programs in the performance evaluation, as measuring the impact of the optimization would be extremely difficult to do in a fair manner. Stream pipelines are used for different reasons and with different workloads, as small parts of bigger applications. Many stream pipelines in existing code are not performance critical; conversely, programmers sometimes avoid using streams exactly for performance reasons, as discussed in the introduction.

The performance measurements are made using the Java Microbenchmarking Harness (JMH) tool [63], a benchmarking tool designed for JVM-based languages included in the OpenJDK project. It performs a series of iterations to warm up the JIT before doing proper testing iterations. In our experiments we perform 5 warm-up iterations and 10 normal iterations, and the presented number is the average over those 10 iterations. We omit confidence intervals, as fluctuations between runs are negligible compared to the differences resulting from the use of optimization and the choices of library and VM [29].

We have performed experiments on the four different Java VMs and versions shown in table 4.1. For each VM we measure the performance of each micro-benchmark before and after optimization. For each benchmark we have four groups. The *Baseline* group constitutes the benchmark implemented with Java `for`-loops, while the *Pull* and *Push* groups use the simple library implementation described in section 4.2. We include our own stream library implementations to show that they suffer from the same performance deficiencies as the Java stream implementation compared to the baseline, and that the optimization can yield performance improvements for both pull- and push style stream APIs. Finally, the *Stream* group uses the stream implementation of the Java standard library. The results can be found in figs. 4.12 to 4.15. The `sum` and `sumOfSquaresEven` benchmarks are shown in fig. 4.9a and fig. 4.1, respectively.

The results show that, when using Java's stream library, after optimization 10 of the 11 benchmarks have comparable performance to that of the baseline implementation.

The technique fails to optimize the stream pipeline for `flatMapTake` in the Stream group on all VMs except Oracle HotSpot VM 8 (indicated by gray bars). This benchmark features a short-circuiting pipeline that includes a `flatMap` operator, which uses a lazily-initialized buffer to hold elements from its generated streams.[13]

[13]All the VMs use the OpenJDK implementation of the standard library for streams. The Oracle

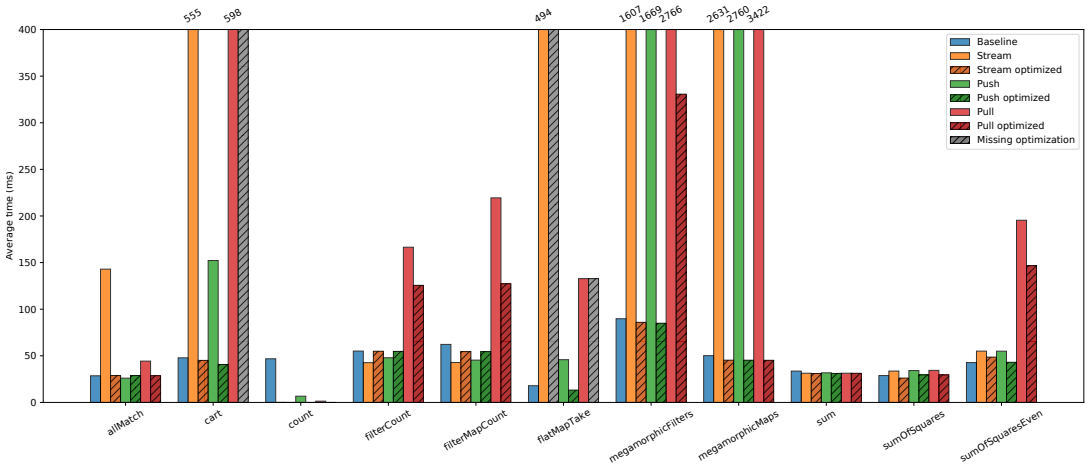


Figure 4.12: Micro-benchmarks run on OpenJDK 13.

The lazy initialization pattern results in too much imprecision causing the analysis to abort. This can be remedied in future work by more precise analysis.

Across all implementations we experience a massive speedup for the (pathological) `cart`, `megamorphicFilters`, and `megamorphicMaps` benchmarks. The `count` benchmarks (which count the number of elements in a stream) experience drastic speedups after optimization, because the JIT can determine that the result is equivalent to the length of the supplied array, effectively making the run-time negligible.

Although our focus is on Java's stream library, we also test the applicability of our approach for pull-style streams. Across all tests, the analysis is too imprecise to optimize the `cart` and `flatMapTake` benchmarks in the Pull group. These benchmarks all use the `flatMap` stream operator. In the pull stream implementation, this operator assumes the iterator protocol in that calls to `get` are preceded by a call to `hasNext` returning `true`. A relational analysis is required to separate the abstract states for when `hasNext` returns `true` or `false` respectively. In many cases the performance of the optimized pull-style stream pipelines does not match that of the baseline, nor the performance of the optimized Java pipelines. This is due to a suboptimal structure of the optimized bytecode, which results in the JIT compiler generating performance-wise worse machine code. The same reason explains how optimized code in some cases performs marginally worse than the unoptimized version, as seen in the `filterCount` and `filterMapCount` benchmarks with OpenJDK 13 in fig. 4.12. Further cleanup transformations are needed to make the bytecode as efficient as the baseline.

There are some differences between the results on the various VMs. The Oracle HotSpot VM is slower for some benchmarks in the Baseline group compared to OpenJDK 13, which is not surprising as the OpenJDK VM has experienced five more years of development. However, we still experience the same relative speedup when the optimization is applied.

Hotspot VM uses an earlier version that does not use lazy initialization.

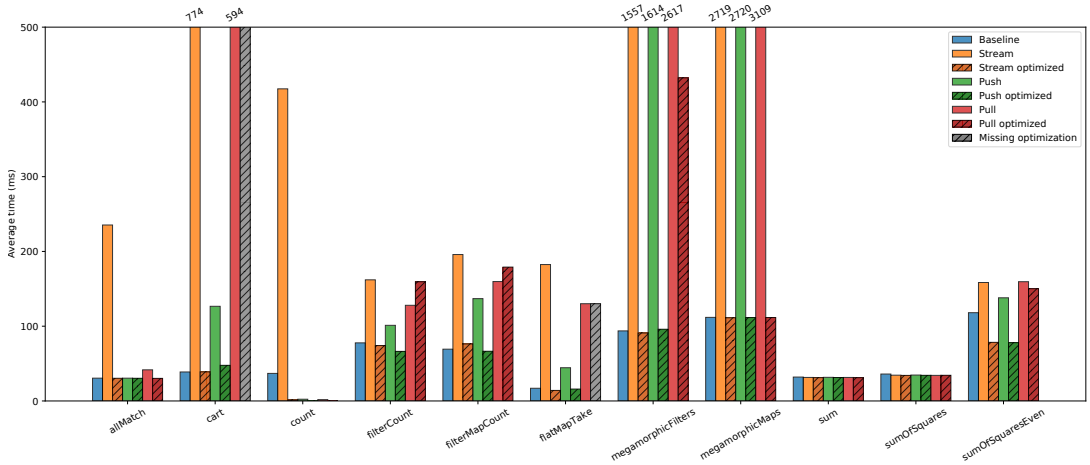


Figure 4.13: Micro-benchmarks run on Oracle's JDK 8.

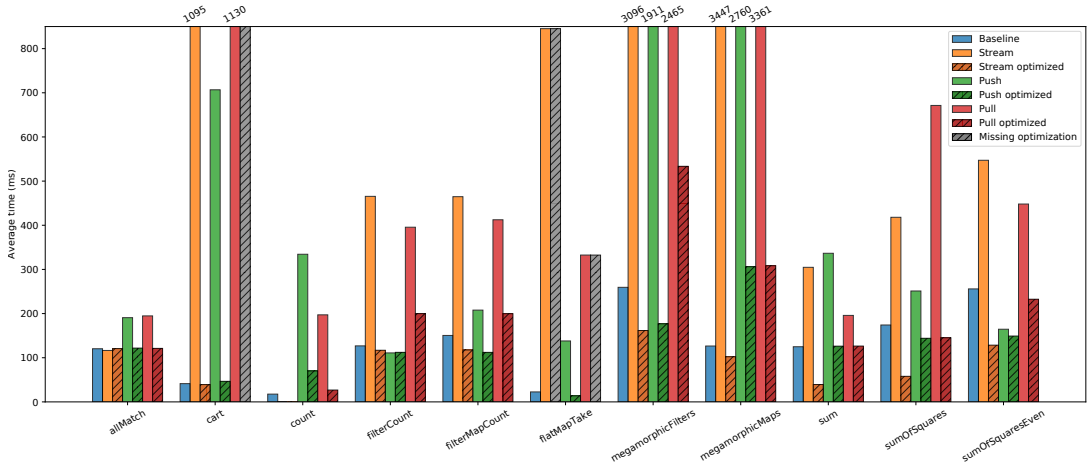


Figure 4.14: Micro-benchmarks run on OpenJ9 13.

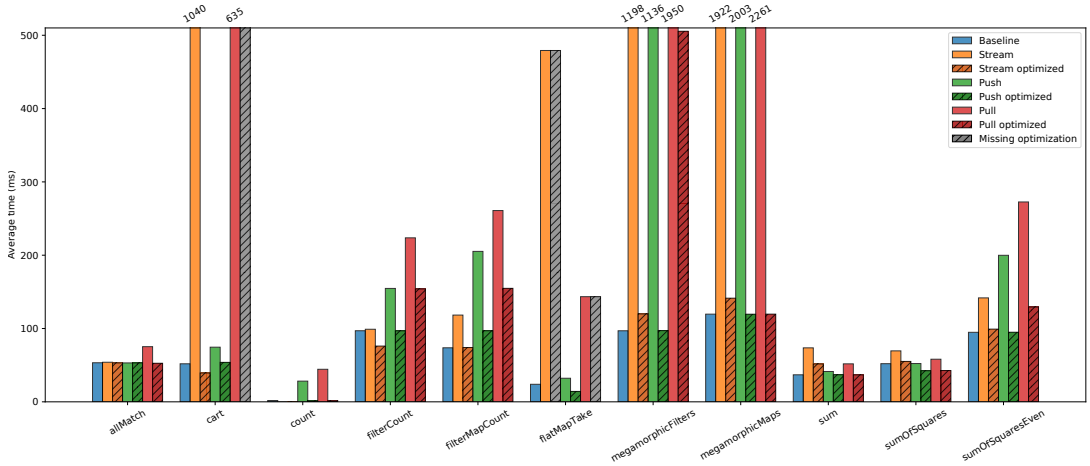


Figure 4.15: Micro-benchmarks run on GraalVM 11.

```java
public int megamorphicMaps() {
  return IntStream.of(v)
          .map(d -> d * 1)
          .map(d -> d * 2)
          .map(d -> d * 3)
          .map(d -> d * 4)
          .map(d -> d * 5)
          .map(d -> d * 6)
          .map(d -> d * 7)
          .sum();
}
```

(a) Benchmark implemented with Java streams.

```java
public int megamorphicMaps() {
  int acc = 0;
  for(int i = 0; i < v.length; i++)
    acc += v[i]*1*2*3*4*5*6*7;
  return acc;
}
```

(b) Benchmark implemented with a for loop.

```java
public int megamorphicMaps() {
  int[] values = v;
  int endExclusive = values.length;
  // bounds and null checking omitted
  int state = 0;
  if(values.length >= endExclusive) {
    int i = 0;
    if (0 < endExclusive) {
      do {
        int t = values[i];
        int t2 = t * 1;
        int t3 = t2 * 2;
        int t4 = t3 * 3;
        int t5 = t4 * 4;
        int t6 = t5 * 5;
        int t7 = t6 * 6;
        int t8 = t7 * 7;
        state += t8;
      } while (++i < endExclusive);
    }
  }
  return state;
}
```

(c) Decompiled benchmark after optimization.

Figure 4.16: The `megamorphicMaps` benchmark.

For the OpenJ9 VM, we also experience significant speedups for the optimized code, although OpenJ9's absolute performance seems to be below that of the other VMs. In some of the benchmarks, in particular the `sum` benchmarks, the Stream optimized code seems to be twice as fast as the baseline, which is suspicious. A plausible explanation for this is that the Java Microbenchmarking Harness is geared towards performance evaluation of HotSpot-based virtual machines (and OpenJ9 is not based on the HotSpot VM), which may lead to inaccurate measurements.

The `megamorphicMaps` benchmark that was highlighted in section 4.1 for its exceptionally poor performance when executed with Java's streams is presented in fig. 4.16. Figures 4.16a and 4.16b show the code that is executed in the Stream and Baseline group, respectively. In fig. 4.16c we show the optimized code after decompilation. The JIT compiler is able to transform both the baseline and optimized code into equally efficient machine code, but it is perhaps not immediately clear to the programmer that the code in fig. 4.16a is semantically equivalent to the code in fig. 4.16c. The optimized code shows similar structure to that of the baseline implemented with a for loop, but includes several transformation artifacts. One such artifact is the check of `values.length >= endExclusive`, which is always true. This could be removed with additional simple intraprocedural cleanup transformations, although it does not affect the performance.

Table 4.2: Results of the evaluation on optimization of stream pipelines in general Java programs.

Category	Count	%
Successful optimization	5 293	77%
Imprecise resolution of call target	985	14%
Use of advanced stream operators	260	4%
Escaping pipeline object	121	2%
Infinite recursion	34	<1%
Other	186	3%
Total	6 879	100%

RQ2: Evaluation on General Programs

To answer the second research question, we run the analysis and transformation on a suite of 28 different Java projects that use streams, to evaluate how many stream pipelines the analysis is able to optimize. Projects are randomly chosen from the RepoReapers dataset [57] under the criteria that we can build the project and that the project contains uses of streams.

Since we are not interested in evaluating the quality of the exact choice of pre-analysis, in this experiment we use a simple alternative to dataflow analysis to decide the stream source types. The ability to optimize a pipeline does not hinge on the concrete type of the stream, only that we know which one it is. For this reason, for streams created from collections (i.e., using the `stream` method in a sub-class of `java.util.Collection`), we simply choose a specific concrete collection type, such as `ArrayList`. For other kinds of stream sources, this pre-analysis simply aborts.

We identified 6 879 sequential stream pipelines in the chosen projects. As explained in section 4.6, whether a stream pipeline can be optimized with our approach is "all or nothing". This gives us a simple way to classify each attempt to optimize a pipeline as being successful or not: If the optimizer succeeds in inlining *all* the stream library code used in the pipeline into the method containing the pipeline, then the optimization is successful. For each pipeline, we invoked the combined analysis and transformation, and recorded whether the pipeline was successfully optimized or not.

In the cases where the optimization is not successful, we have attempted to identify the most likely cause. The results of the experiment are presented in table 4.2. Out of 6 879 pipelines, 5 293 (77%) are successfully optimized. This leaves 1 586 pipelines that fail to optimize for different reasons. The most prominent reason is that the analysis aborts due to imprecise type information at call sites, making it impossible to statically track the interprocedural control flow of the stream pipeline. This imprecision can arise from different sources as described in section 4.5. One is that the simple pre-analysis implementation fails to deliver the type information needed to analyze the construction of the stream source, which accounts for about half of the 985 cases. (That may happen if the stream is created neither from a

collection nor from static methods such as `IntStream.of`.) Incorporating a full-fledged pointer analysis, such as Boomerang, [77] can likely help the analysis in these cases. The analysis also experiences imprecision when the pipeline structure depends on branching (for example when an intermediate operation is applied to a stream only under some conditions, as in line 162). More advanced techniques could insert optimized code for both cases and branch on the original condition. The next most common cause of inability to optimize is the use of stream operators that the analysis is not precise enough to handle. This includes the `LongStream.range` source, which leads to infinite recursion, the `toArray` and `concat` operators, and the `flatMap` operator when involved in a short-circuiting pipeline as outlined in section 4.8. These operators include some complex state that is initialized during pipeline execution which the analysis is unable to follow. This can cause the analysis to abort due to an imprecise resolution of a call target, as described in section 4.5, or make the analysis result too imprecise to allow meaningful optimization, as described in section 4.6. In 121 cases, the main analysis aborts due to an object escaping the analyzed part of the code, and in 34 cases the analysis aborts due to uncontrolled growth of call strings. Both of these conditions are described in section 4.5. The remaining unsuccessful cases are harder to classify. In most of these cases the analysis succeeds but Java's access control mechanisms prevent optimization, as discussed in section 4.6.

In summary, the results from this experiment show that the relatively simple static analysis presented in section 4.5 can produce the information needed to optimize stream pipelines in a variety of programs. Moreover, the technique is quite cheap to apply. In our experiments, the analysis and transformation take approximately one second to apply for each pipeline on average.

4.9 Related Work

Most work on compiler optimization for Java focuses on JIT optimizations [5, 6], and there is (surprisingly) little work on AOT optimizations in general for Java and related languages. To our knowledge, we are the first to investigate the use of AOT optimization for eliminating the massive abstraction overhead of Java stream pipelines.

The bytecode-to-bytecode optimizer described by Budimlic and Kennedy [12, 13] applies a transformation called object inlining, which inlines all data and code from selected objects, much like our use of method inlining and stack allocation, however, they do not present any strategies for when to apply the transformation. They also encounter the limitation of inlining methods that access private members. The term 'object inlining' has also been used for another kind of optimization that fuses together objects to reduce the number of object allocations, without method inlining or stack allocation [27].

Another related technique is the interprocedural escape analysis for guiding stack allocation optimization for Java by Choi et al. [17]. Our dataflow analysis (section 4.5) performs a variant of escape analysis by the use of the unrelated lattice element, to determine which objects may escape the stream pipeline code.

The Interflow optimizer [73] for Scala Native uses a combination of flow-sensitive type inference, method duplication, partial evaluation, partial escape analysis, and inlining. It focuses on optimizations for Scala's collection library, not for stream pipelines, and is designed for native code generation instead of bytecode-to-bytecode transformation. By targeting native code, they avoid the problems with Java's access modifiers discussed in section 4.6. On the other hand, by choosing bytecode-to-bytecode transformation, our approach is easier to incorporate into existing build processes and execution platforms.

Our approach builds on ideas from the techniques mentioned above, and applies them to optimize stream pipelines. By focusing analysis and transformation on stream pipeline code that has large potential for optimization, we can afford more expensive analysis than the general purpose optimization techniques.

Our optimization technique can also be viewed as a form of program specialization [72], where we specialize the stream library code to each individual stream pipeline. Instead of using a binding-time analysis as in traditional partial evaluation, we use a specialized analysis that simultaneously infers types and points-to information to guide the transformations.

Khatchadourian et al. [36] have developed a tool for optimizing Java streams that uses a static typestate analysis to determine whether it is advantageous to convert a sequential stream to a parallel one or vice versa. Parallel computation is a natural source of performance improvement, so their goal is to determine preconditions for when it is safe to execute pipelines concurrently. While parallel streams can offer better performance, it does not address the inherent overhead that is currently present when using Java's streams sequentially, as discussed in the introduction.

Declarative data processing has close ties to functional programming. Deforestation [90] is a technique that transforms functional programs that operate on trees (in particular lists) into equivalent programs without allocating intermediate results in new trees, thereby improving run-time performance. Many variants of deforestation exist, but mostly for functional programming languages. These techniques are difficult to adapt to optimize code that uses Java's stream library, in particular because of its advanced features described in section 4.2.

For programming languages with advanced meta-programming capabilities, such as staging, efficient stream implementations can be obtained by implementing stream fusion and other optimizations within the libraries themselves. The *strymonas* library for Scala and OCaml [39], ScalaBlitz for Scala [67], the fold-based fusion technique for Scala by Jonnalagedda and Stucki [33], and LinqOptimizer for C# and F# [65] follow that approach. These techniques cannot be adapted to Java, because it lacks the necessary language features. Also, our goal is to enable optimization for Java's existing stream library, not to replace it.

The stream library for Java by Biboudis et al. [9] aims for extensibility, not to reach the performance of imperative code.

C# supports declarative data processing in the form of Language-Integrated Query (LINQ), which suffers from similar performance problems as Java streams compared to hand-optimized code. The Steno tool [58] makes it possible to translate declarative

LINQ queries into imperative code, using iterator fusion and nested loop generation optimizations. Earlier work has applied similar approaches as Steno for Common Lisp and Pascal programs [91]. The key difference to our technique is that Steno relies on hardwired knowledge about the semantics of all the available LINQ operators and thus does not need to look at their implementations; in contrast, our approach is not limited to a specific API but instead relies on static analysis of the stream library implementation.

Also for C#, Adamus et al. [1] have developed a technique for optimizing LINQ queries by identifying free expressions in nested queries. By lifting these expressions out of the nested query they can avoid redundant re-computation at run-time, thus improving performance. Their technique builds on the idea of rewriting stream pipelines, and is not concerned with the performance overhead of using LINQ queries compared to hand-optimized code.

4.10 Conclusion

Streams are a powerful abstraction mechanism in Java programming, but they incur a large performance overhead, which JIT optimization has been unable to mitigate. In this work we exploit the fact that stream pipelines are relatively small pieces of code, which makes them amenable to high-precision interprocedural analysis and optimization. We have demonstrated the feasibility of AOT optimization of Java stream pipelines. By aggressively applying method inlining and stack allocation transformations driven by a static type/pointer analysis, our experimental results show that a variety of stream pipelines can be automatically transformed into efficient imperative-style code that has much better performance characteristics. For 10 of 11 micro-benchmarks, the resulting bytecode is as effective as hand-written imperative-style code, and 77% of 6879 stream pipelines found in real-world Java programs are optimized successfully. Since the optimizer is fast (even for a prototype implementation) and structured as a bytecode-to-bytecode transformer, it is easily deployed in ordinary build processes. Moreover, the approach is not restricted to Java's push-style stream implementation but also produces good results for a simple pull-style library.

The experimental results also identify opportunities for future work. Most importantly, more pipelines could be optimized if the analysis is improved to be able to reason more accurately about short-circuiting operations. Also, incorporating relational analysis can lead to improved precision necessary for optimizing certain operations when using a pull-style stream library.

4.11 *Epilogue:* Pre-analysis in Practice

In the evaluation of STREAMLINER we used an unsound and optimistic pre-analysis. Recall from section 4.8 that for queries for the type of receivers of calls to `java.util.Collection<T>.stream()` it always answers with an arbitrary concrete collection

type that is valid according to the type information present in method. For the purpose of measuring how many pipelines fail to optimize due to limitations experienced in the main analysis phase (as opposed to failures directly caused by imprecise pre-analysis), the optimistic pre-analysis was a reasonable choice. Since STREAMLINER performs program optimizations, it is critical that the different involved analyses are sound, otherwise the optimizations may not preserve program behavior. Therefore this pre-analysis is unsuitable for use in "real" applications of the tool. Section 4.4 lists several pointer analysis tools that are both designed to be (mostly) sound and can (in principle) answer the queries that STREAMLINER asks. A natural question to ask is how well STREAMLINER works when it is paired with one of those off-the-shelf pointer analyses, i.e., how many of the 6879 pipelines studied for **RQ2** can we optimize with a sound pre-analysis based on state-of-the-art tools. In this section we present a few state-of-the-art analysis tools and describe how they can be used to answer STREAMLINER queries. We then experimentally verify how well our tool works when coupled with these pointer analyses by integrating them into the experimental setup originally used to answer **RQ2**.

There are two different modes of operation for the pointer analyses we will investigate. The first mode is whole-program pointer analysis. In a whole-program pointer analysis the program we need to answer queries for is analyzed once up-front. The results of the analysis are consulted when a type query is received from STREAMLINER. These queries are answered quickly because no more analysis is required. The second mode of operation is demand-driven pointer analysis. In a demand-driven pointer analysis the main work of the analysis is postponed until a query is received from STREAMLINER. The analysis may require some additional up-front analysis before queries can be answered, but this can be cheaper than traditional whole-program pointer analysis. Demand-driven pointer analysis is practical because the work of the analysis can be tailored to the individual queries. It may be possible to avoid a lot of the work that whole-program analyses do for parts of the program that are irrelevant with respect to the queries. However, the implementation of efficient demand-driven pointer analyses is difficult and generally requires a different set of tools than those used for the implementation of whole-program analyses.

WALA [28] is the first tool that we will investigate. WALA contains a framework for performing analysis of Java programs, on top of which variants of Andersen's pointer analysis [3] are implemented. WALA also includes an implementation of the demand-driven pointer analysis by Sridharan and Bodík [79]. These analyses have a plethora of options that can be tuned, so we are glad to have Manu Shridharan — one of the core developers of WALA and author of the demand-driven analysis — help us select sensible configurations for our use case. This results in three different pre-analyses based on WALA: 1) a pre-analysis that answers queries using WALA's demand-driven pointer analysis seeded with an imprecise but quick-to-construct call graph built with Class Hierarchy Analysis *(DemandCHA)*, 2) a pre-analysis that answers queries based on the results of a whole-program pointer analysis *(OfflineCFA)*, and 3) a pre-analysis that combines the two previous approaches, but seeds the demand-driven pointer analysis with a (potentially more precise) call graph computed by the

Table 4.3: Optimization results & query statistics for pre-analyses.

	Initialization		Queries		
Pre-analysis	Success	Time	Success / Total	Avg. time	Opt. pipelines
Baseline	28	0.0 h	0 / 6351 (0.0%)	0 ms	1108 (16.1%)
Optimistic	28	0.0 h	13137 / 24682 (53.2%)	0 ms	5218 (75.9%)
WALA:					
DemandCHA	27	3.6 h	2512 / 7393 (34.0%)	627 ms	1796 (26.1%)
OfflineCFA	25	6.9 h	1830 / 6715 (27.3%)	1 ms	1475 (21.4%)
Combined	25	7.0 h	1830 / 6715 (27.3%)	57 ms	1475 (21.4%)
Soot:					
SPARK	25	1.2 h	1749 / 7202 (24.3%)	0 ms	1481 (21.5%)
Boomerang	25	1.3 h	1819 / 7272 (25.0%)	5088 ms	1481 (21.5%)

whole-program analysis *(Combined)*. The whole-program analysis is configured to be context-insensitive, it unsoundly ignores reflection, and it uses a heap abstraction that represents all concrete objects of the same type with a single abstract object. Although this is imprecise, the analysis does not scale to our benchmarks with more precise configurations.

The second tool we will investigate is Soot [85] which contains the SPARK whole-program pointer analysis framework [45]. With the default SPARK configuration, which defines a subset-based pointer analysis akin to Andersen's analysis, we create a pre-analysis that answers queries in the same way as *OfflineCFA*. We also define a pre-analysis based on Boomerang [77], a demand-driven pointer analysis technique implemented on top of Soot. Specifically we use the implementation of Boomerang in the framework of Synchronized Pushdown Systems [78]. Like the demand-driven pointer analysis implemented on top of WALA, this one must also be seeded with a call graph. For this purpose we use the call graph computed by SPARK and, similar to the *Combined* pre-analysis described above, we only invoke Boomerang when the SPARK results (which are readily available after computing the call graph) are too imprecise to answer a query. In the original evaluation of Boomerang a query timeout of 1 second is used. For our evaluation we use a timeout of 15 seconds, which we think is fair, but otherwise use Boomerang's default configuration.

Finally we include a *Baseline* pre-analysis that purposefully fails to answer any queries and the *Optimistic* pre-analysis used in the original **RQ2** experiment. The *Baseline* pre-analysis makes it possible to measure how many additional stream pipelines can be optimized when STREAMLINER is paired with one of the realistic pre-analyses.

For each of the pre-analyses we repeat the **RQ2** experiment and record additional statistics about the performance of the pre-analysis. The results of these experiments are available in table 4.3. The second and third columns of the table contain metrics related to initialization of the pre-analysis itself. The *Success* column denotes the

number of projects initialization succeeded for (of which there are 28 in total). For the WALA-based pre-analyses, initialization fails for one project because it targets a newer version of Java that is unsupported by WALA. The WALA whole program analysis — used in the *OfflineCFA* and *Combined* pre-analyses — fails in two instances due to (perceived) inconsistencies in class files compiled from Groovy, a programming language used with the Gradle build system that also targets the JVM. When initialization fails, the pre-analysis cannot answer any queries from STREAMLINER, and the received queries are not included in data used to derive the query metrics. The SPARK pre-analyses fail to initialize for the same two projects as the ones where WALA had Groovy-related issues. It also fails to initialize for one project due to multiple incompatible definitions of the same class. The *Time* column contains the total time spent on initialization over all projects. The results show that the WALA demand-driven pointer analysis does require less initialization time compared to the WALA whole-program analysis, but also that the time saved is not enough to make the pre-analysis applicable in scenarios that require rapid feedback (at least not without further optimizations).

The next two columns contain metrics related to the individual queries processed by the pre-analysis. The first of them contains the number of queries answered successfully, the total number of received queries, and the derived success rate. Due to design decisions in the implementation of STREAMLINER, a type query is successful when it is answered with a single concrete type. There are many scenarios where such an answer is impossible (i.e. the result would be unsound), so a success rate of 100% is generally unattainable. Potential changes that can make STREAMLINER accept less precise answers are discussed in section 4.8 and below. Notice that the total number of queries received is not the same across the pre-analyses. One reason is that, as explained above, when initialization fails, the pre-analysis cannot answer any queries for that project. However, the discrepancy between the number of received queries for the *Baseline* and *Optimistic* pre-analyses, which cannot have failed initialization, is large. This is because STREAMLINER can ask multiple queries during the main analysis phase. If the main analysis phase aborts early due to failed queries, potential follow-up queries will not be performed. The second of the query metric columns contains the average processing time of each query. For the pre-analyses based on WALA, we see that the purely demand-driven pointer analysis approach (*DemandCHA*) spends more time per query than the ones based on whole-program analysis, but it still spend less total time overall when initialization time is included (while also answering more queries in total). This suggests that demand-driven pointer analyses can be a good fit for our approach, at least with respect to performance. Interestingly, the results show that the combination of a demand-driven pointer analysis and a whole-program analysis has negligible impact on the success rate (compare *OfflineCFA* with *Combined* & *SPARK* with *Boomerang*). We expected the demand-driven pointer analyses to be able to recover some of the precision that is sacrificed to make the whole-program analyses scalable, but this was not the case for our benchmarks. Both the demand-driven analysis in WALA and Boomerang time out for most queries that the whole-program analyses cannot answer. We tried to increase

the budgets for both analyses by an order of magnitude, but this had almost no impact on the number of timeouts.

The final column contains the number of pipelines that can be optimized based on the type query answers provided by the pre-analysis. With the *Baseline* pre-analysis STREAMLINER can optimize 1108 pipelines. When paired with a pre-analysis based on off-the-shelf pointer analysis, the number of optimizable pipelines is between 1475 and 1796, which is an increase of $30-60\%$. Interestingly, the difference in the results for pre-analyses based on whole-program analysis between WALA and Soot is negligible. The main result is that the pre-analyses based on state-of-the-art tools allow for a significant increase in optimizable stream pipelines, but also that there is still room for improvement when we compare with the *Optimistic* pre-analysis.[14] This improvement can come in the form of better off-the-shelf analyses, but also from changes to our approach that can allow optimization to take place with less precise query results.

Sound pre-analysis

The pointer analysis tools presented in the previous section are designed to be mostly sound (soundy [49]), which is fine for many applications of pointer analysis, such as code navigation in an IDE and bug detection. However, STREAMLINER optimizes the analyzed programs based on information gained (in part) from the pre-analysis. The analyses and optimizations used in our technique are sound, i.e. they preserve program behavior under the assumption that the results provided by the pre-analysis are sound. But if the pre-analysis fails to abstractly model some program behaviors that can occur at run-time, STREAMLINER's optimizations may change the behavior of the program and even introduce bugs.

Designing fully sound and useful pointer analyses is a difficult problem. Consider the example method `sum` in fig. 4.17. To optimize the pipeline in this method, STREAMLINER needs to know the concrete type of the collection passed to `sum` as a parameter. The performed optimization preserves program behavior if the argument has the same concrete type at *all* call sites. However, in the presence of opaque code due to reflection and classes loaded at run-time, it is really impossible to statically guarantee anything for *all* call sites, as there may be call sites that are unknown at compile-time. Uses of reflection and other dynamic features of the Java language are widespread in practice [42], so simply ignoring them is unsatisfactory. For the `sum` method in fig. 4.17 the consequence is that, even though a pre-analysis might be able to determine that only a single concrete type is possible at all observable call sites, it is still problematic to optimize for that type alone. Recent work by Smaragdakis and Kastrinis [75] on defensive pointer analysis shows that an analysis designed to only compute points-to information for variables whose values are guaranteed to only depend on information in the current calling context can be both scalable and precise in practice, even with a high degree of context-sensitivity. In their experiments the

[14] There is a slight deviation in the number of optimized pipelines compared to the results in the original **RQ2** experiment due to minor changes in test setup and the STREAMLINER implementation.

```
251 public class Example {
252   private static int sum(Collection<Integer> c) {
253     return c.stream().mapToInt().sum();
254   }
255
256   public static void fun() {
257     System.out.println(
258       sum(List.of(1, 2, 3)) +
259       sum(Set.of(4, 5, 6)));
260   }
261 }
```

Figure 4.17: A method that sums the integers in a collection with a stream

defensive pointer analysis can compute sound points-to information for 45.6% of all variables in some calling context, and for 35.5% of variables specifically in the empty context. A devirtualization client is also studied. This client wants the set of callees for a virtual call to be a singleton set, which is very similar to our pre-analyses. On their benchmarks the results show that the defensive analysis satisfies the devirtualization client in 38.7% of cases (when restricted to devirtualization in the empty context), whereas a soundy state-of-the-art pointer analysis can satisfy the client in 89.3% of all cases. This is disappointing wrt. STREAMLINER, which already fails to optimize most pipelines when paired with soundy analyses.

One way to circumvent the issue with optimization based on unsound pre-analyses, and also make optimization possible when the pre-analyses are too imprecise, is to insert guards around the optimized code that assert that the run-time type of the argument is as expected. If the assertion fails the method can fall back to the unoptimized version of the code and preserve its expected behavior. A different approach is to attempt to recursively inline the method containing the stream pipeline into its callers until the concrete type of the `.stream()` receiver no longer depends on information from callees. In the `fun` method of the example class in fig. 4.17, there are two invocations of `sum` with different concrete argument types. However, if the calls are inlined in both cases, the optimizer can trivially specialize the code for both types that are now known in the context of `fun`. Both of these approaches have the potential to increase the code size of the program after optimization, so implementing them, verifying that they enable more optimizations, and checking whether the optimizations are worthwhile compared to the potential increase in code size, is an opportunity for future work.

Chapter 5

Detecting Blocking Errors in Go Programs using Localized Abstract Interpretation

By Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Published in the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 2022. Sections 5.7 and 5.8 are new material.

Abstract

Channel-based concurrency is a widely used alternative to shared-memory concurrency but is difficult to use correctly. Common programming errors may result in blocked threads that wait indefinitely. Recent work exposes this as a considerable problem in Go programs and shows that many such errors can be detected automatically using SMT encoding and dynamic analysis techniques.

In this paper, we present an alternative approach to detect such errors based on abstract interpretation. To curb the large state spaces of real-world multi-threaded programs, our static program analysis leverages standard pre-analyses to divide the given program into individually analyzable fragments. Experimental results on 6 large real-world Go programs show that the abstract interpretation achieves good scalability and finds 104 blocking errors that are missed by the state-of-the-art tool GCatch.

5.1 Introduction

The key feature of the Go programming language is its channel-based approach to concurrency with lightweight threads. Instead of communicating via shared memory, Go advocates the use of channels to avoid errors involving data races. This language design choice is a central reason for the popularity of the language. More than 270 000 open source projects on GitHub use Go, including prominent applications, such as,

Docker and Kubernetes. However, using channels does not eliminate all concurrency-related errors. Previous work has shown that channel-related concurrency errors are frequent in Go programs [47, 48, 84]. A common erroneous pattern involves a thread waiting on a blocking channel operation that will never be unblocked by any other thread, often due to some unexpected condition occurring for potential communication partners. Repeated occurrences of such progress failures may drain system resources and eventually cause execution to halt.

To detect such blocking errors automatically ahead of program execution, we propose an approach based on abstract interpretation. Our analysis first locates channel creation and communication operations by leveraging existing basic analyses for producing call graphs and aliasing information, and identifies for each channel a program fragment covering its operations. Each fragment is then analyzed separately using abstract interpretation, hence the term *localized abstract interpretation*, to infer a finite transition system that flow-sensitively models the state space for the relevant threads and channels in the fragment. Dynamic thread creation may lead to a statically unbounded number of threads, causing challenges for static analysis. The localized analysis approach partially circumvents this issue for fragments that may execute an unbounded number of times at run-time but where each execution only involves a bounded number of threads. The last step consists of analyzing the generated transition systems, looking for configurations where a thread may be blocked indefinitely, in which case a potential error is reported.

Our approach is inspired by GCatch [47], which aims to detect the same category of errors and similarly analyzes selected program fragments one-by-one. The central difference is that GCatch does not use abstract interpretation but instead enumerates potential execution paths for each thread within the fragment and uses an SMT solver to compose the paths and detect blocked threads. We believe it is simpler to model channel operations and other Go language constructs using abstract interpretation instead of SMT encodings. The tool Gomela [25, 26] has similar goals and is based on bounded model checking of Promela-encoded Go programs. It obtains high scalability on real-world code, but detects relatively few blocking errors. Another recent approach is GFuzz [48], which detects blocking errors by fuzzing concrete executions obtained by running the programs' test suites. That approach relies on high-coverage test suites to be effective.

Preliminary experiments show that our localized abstract interpretation approach scales to large Go programs (typically analyzing each fragment in less than a second), it is capable of finding many blocking errors that are missed by the existing state-of-the-art tools, and it has an acceptable false positive rate (less than 50%).

These results are enabled by some pragmatic design choices: (1) A common programming pattern involves worker pools with correlated loops, which we handle in a light-weight manner by modeling only a fixed number of loop iterations. (2) The localized abstract interpreter models a coarse-grained thread scheduler that intuitively ignores potential shared-memory data races, giving a substantial state space reduction. As our goal is not to detect data races but channel-related issues, it is acceptable that not all possible interleavings are explored. Disabling these techniques reduces the

performance and accuracy of the analysis.

In summary, the contributions of this paper are:

- We demonstrate that localized abstract interpretation is a promising approach to detect channel-related concurrency errors. This is notable, because abstract interpretation is historically rarely used for reasoning about such concurrency errors due to the large state spaces that often appear.
- By experimentally evaluating the approach on 6 large real-world Go programs, we show that it compares favorably with the state-of-the-art tools GCatch and GFuzz. Specifically, it detects 104 blocking errors that GCatch misses and 84 that GFuzz misses (thereof 76 missed by both), and most program fragments are analyzed in seconds. Additionally, we show that the pragmatic design choices are important for the efficacy.
- We report on typical scenarios that cause false positives and false negatives, which suggests interesting opportunities for future work.

The approach is implemented in the tool GOAT.[1]

5.2 Background

Go is a statically-typed concurrent imperative language geared towards systems development. Threads (called "goroutines") can be created dynamically like in many other languages. Although Go supports traditional shared-memory communication among threads using locks and other basic synchronization mechanisms, its hallmark concurrency feature is channels inspired by Hoare's CSP. A channel is a bounded queue that can be accessed by multiple threads. Reading from a channel blocks until data is available, and sending to a channel blocks if the channel is full. The channel capacity is selected when the channel is created. For example, `make(chan T, 3)` creates a channel for values of type `T` with capacity 3. Channels with capacity zero are called *synchronous* because they require reads and corresponding writes to happen simultaneously. Alongside the send and receive operations (c <- $\cdots$ denotes writing to a channel c and $\cdots$ <- c denotes reading), Go's `select` statement allows nondeterministic choice among enabled channel operations. Channels may also be closed, at which point their blocked receive operations are unblocked and further send or close operations will fail.

Channel-based concurrency is a powerful and popular language mechanism that prevents low-level data races, but it does not prevent all concurrency issues. If *all* threads are blocked on some channel operations, waiting for other threads to send or receive, Go's built-in deadlock detector aborts execution. However, a much more common situation is that some but not all the threads are blocked, waiting for channel operations that can never occur, because some thread has taken an execution path

[1] **Go a**nalysis **t**ool

that was not anticipated by the programmer. This situation may violate desirable progress properties of the program, and it causes "goroutine leaks" that consume precious system resources. The goal of our work is to automatically detect whether such blocking errors are possible in a given Go program.

Figure 5.1 illustrates such an error, found in *etcd*,[2] a distributed key-value store implemented in Go. It involves creating a configurable server (line 44), and waiting until it is ready (line 45). Servers implement the `Server` interface (line 1), which contains a `Ready` method that returns a channel. Since creating a server may be asynchronous, a rendezvous point is established by reading from this channel. The servers produced by `NewServer` (line 6) are represented by the `server` structure (line 23), where the `Ready` method (line 35) returns the channel embedded in the `readyc` field. Servers are ready when the `readyc` channel is closed (line 31). `NewServer` normally achieves this by spawning a thread (expressed using the **`go`** keyword), **`go`** `s.listenAndServe()` (line 19), that executes the **`close`** operation. However, if an error occurs (line 14), the function returns without closing the channel. In this case, reading from the `readyc` channel will block indefinitely. The proposed fix for this blocking error is to read from both the `readyc` and `errc` channels simultaneously using a **`select`** statement (lines 45–49).

This example illustrates how channels may easily be misused. Any client using the API similarly to the `testServer` function in fig. 5.1 exposes itself to the resource leak. Fixing the error requires knowledge of the implementation of `NewServer` and `Ready`, which is complicated by having the implementation of `*server` methods hidden behind the `Server` interface.

This particular error is missed by the existing tools GCatch and GFuzz. GCatch fails to detect the erroneous execution path through the relevant part of the program. GFuzz performs fuzzing of program executions by re-ordering choices of case clauses in **`select`** statements, which does not suffice to discover that `err` may be non-**`nil`** in this case. In contrast, GOAT successfully discovers the error by analyzing a program fragment containing the functions that involve operations on the `readyc` channel. The fragment does not contain the function `net.Listen`, so the analysis makes a worst-case assumption about the possible value of `err`, therefore considering the path with the server start-up error where no thread is spawned at line 19. The developers of *etcd* have subsequently confirmed the error and approved the proposed fix.

5.3 Approach

Reasoning automatically about the presence or absence of blocking errors in Go programs requires flow-sensitive analysis of multi-threaded code and channel state. Our approach to obtain scalability to large, real-world programs is to consider each syntactic channel creation site individually (e.g., line 9 in fig. 5.1) and ignore program code that is unlikely to affect whether operations on channels created there may block.

[2] `https://etcd.io/`

```
1   type Server interface { // Server API
2     Error() chan error
3     Ready() chan struct
4   }
5
6   func NewServer(cfg) Server {
7     s := &server{ // Server object
8       ...
9       readyc: make(chan struct{})  // Ready channel
10      errc:   make(chan error, 16) // Error channel
11    }
12    ...
13    _, err = net.Listen(s.from.Scheme, addr)
14    if err != nil { // Server startup error
15      s.errc <- err
16      s.Close()
17      return s
18    }
19    go s.listenAndServe()
20    return s
21  }
22
23  type server struct { // Server data structure
24    ...
25    readyc chan struct{}
26    errc   chan error
27  }
28
29  func (s *server) listenAndServe() {
30    ...
31    close(s.readyc) // Close ready channel
32    for { ... } // Listen-and-serve loop
33  }
34
35  func (s *server) Ready() { // Ready channel getter
36    return s.readyc
37  }
38  func (s *server) Error() { // Error channel getter
39    return s.errc
40  }
41
42  func testServer() {
43    ...
44    s := NewServer(cfg)
45-   <-s.Ready() // Potential blocking error
46+   select {
47+     case <-s.Ready(): ... // Proceed normally
48+     case err := <-s.Error(): ... // Handle error
49+   }
50    ...
51  }
```

Figure 5.1: A blocking error in *etcd*. (Irrelevant details have been elided, and explanatory comments have been added.)

As an example, the code involving the blocking error of the `readyc` channel, and the suggested fix shown in fig. 5.1 is a tiny fraction of the 180 KLOC that constitute *etcd*.

Overall, the analysis of a given Go program is divided into three main phases:

1. The **pre-analysis** phase identifies channel creation sites syntactically, and for each of them selects a *program fragment* consisting of functions that likely cover the relevant channel operations. To this end, we leverage the existing Go parser and Andersen-style points-to analysis [82]. The selection of program fragments is inspired by the one used by GCatch [47] as explained in detail in section 5.3.

2. The main work is performed in the **abstract interpretation** phase that analyzes each program fragment and builds a *superlocation graph*, which is a finite transition system that models the state space of the fragment. In section 5.3 we describe the abstract domain and abstract semantics of this analysis, and how it handles interactions with program code outside the fragment. These abstractions are carefully designed to track enough information to enable reasoning about blocking channel operations, while allowing typical fragments to be analyzed in less than a second.

3. In the **blocking error detection** phase described in section 5.3, the superlocation graphs are traversed, searching for abstract threads at channel operations with no possible unblocking path. Such threads may likely block indefinitely at run-time.

We conjecture that most uses of channels in real-world Go programs are amenable to such localized analysis. By bounding the analysis time for each program fragment, we effectively obtain an analysis technique that scales linearly in the size of the program (if ignoring the time for the pre-analysis). For programs with multiple entry points, the precision of the pre-analysis can be improved by running it separately for each entry point, essentially treating each entry point as a separate program.

Pre-Analysis

The pre-analysis first runs the ***points-to analysis*** available in the `pointer` package developed by the Go team [82]. Since channels are first-class values in Go, the resulting points-to information [3] tells us for each channel creation site which channel operations may involve channels created at that site. Continuing the *etcd* example from fig. 5.1, channels created at line 9 may be used only at lines 31 and 45 (before the fix is applied). We henceforth identify channels by their syntactic creation site in the program. The points-to analysis result also includes a call graph that approximates which functions may be called at each call site.[3] To support the next steps, we compute

[3]Imprecision of this existing analysis can lead to a large number of callees for some call sites, especially where interface method invocation is involved. To prevent such situations from affecting the main analysis, we prune the call graph at sites where the number of callees exceeds an arbitrarily limit (10 in our experiments). This pragmatic choice increases precision and speed, at the cost of causing unsoundness in the analysis.

the strongly connected components (SCCs) of the call graph and organize them in reverse topological order.

With these initial steps in place, we can perform ***fragment construction*** that selects a set of functions, called a *fragment*, for each channel c in the program. This process is inspired by GCatch [47] but has some extensions as explained below. The basic idea is to locate functions that are likely relevant for reasoning about operations on c, building on the points-to and call graph information. Since this may involve other channels, we first identify a set of likely relevant channels, $\mathscr{P}(c)$ (called P_{set} in [47]). Intuitively, communication operations on channels not in $\mathscr{P}(c)$ are ignored by the abstract interpretation of the fragment constructed for c. The set $\mathscr{P}(c)$ consists of the channel c itself and any other channel c' that satisfies one or both of the following conditions:

C1: c' and c are mutually dependent. Channel c depends on c' if an operation on c that may unblock another operation on c is intra-procedurally reachable from a blocking operation on c'.

C2: c' and c are used in different `case`s of the same `select` statement.

Figure 5.2a illustrates C1 by a blocking error that is due to a communication mismatch, which would not be revealed by analyzing channels individually. We have that `a` depends on `b` because the read operation on line 55, which may unblock the write on line 57, intra-procedurally requires reading from `b` on line 54. Conversely, `b` depends on `a` because writing to `b` on line 58 requires writing to `a` first on line 57. Restricting to intra-procedural reachability is a heuristic that prevents call graph imprecision to lead to very large $\mathscr{P}(c)$ sets.

Figure 5.2b motivates C2 by showing an example of a blocking `select` that would not be detected if $c' \notin \mathscr{P}(c)$ and $c \notin \mathscr{P}(c')$.

An additional condition for both C1 and C2 is that the dominator of c' is reachable from the dominator of c in the call graph. Here, the *dominator* of a channel is the dominator in the call graph of all the functions that might create, use, or return the channel. This additional condition helps limiting the sizes of the $\mathscr{P}(c)$ sets, thereby providing a more fine-grained analysis of the program.

We extend the definition of $\mathscr{P}(c)$ beyond the P_{set} construction of GCatch by also including any channel c' in $\mathscr{P}(c)$ that satisfies the following condition:

C3: c' might carry c as a payload, irrespective of how their dominators are related.

This addition of C3 is owed to our empirical observation of programming patterns involving channels with channel payloads. Figure 5.2c illustrates a non-blocking example where `a` would not be included in $\mathscr{P}(\texttt{b})$ by GCatch, leading the analysis to lose track of the connection between `b` and the payload of `a` when ignoring operations on `a`. Including `a` in the set of channels relevant to `b` allows precise analysis of all communication in the fragment.

```
52  a, b := make(chan int), make(chan int)
53  go func() {
54    <-b
55    <-a
56  }()
57  a <- 1
58  b <- 2
```

(a) Blocking error resulting from channel communication mismatch, captured by condition C1.

```
59  a, b := make(chan int), make(chan int)
60  select {
61    case <-a:
62    case <-b:
63  }
```

(b) Blocking select statement, captured by condition C2.

```
64  a := make(chan chan int, 1)
65  go func() {
66    b := make(chan int)
67    a <- b
68    b <- 3
69  }()
70  <-<-a
```

(c) Common non-blocking communication pattern involving channels with channels as payload, captured by condition C3.

Figure 5.2: Motivating examples for constructing $\mathscr{P}(c)$.

For each set $\mathscr{P}(c)$ we now define a program fragment $\mathscr{F}(c)$ containing each function that for some $c' \in \mathscr{P}(c)$ either creates, returns, or performs a communication operation on c'. Additionally, $\mathscr{F}(c)$ includes all ancestors in the call graph, up to the dominator of those functions. We denote this dominator as the *fragment entry*. Note that $\mathscr{F}(c)$ typically does *not* include all functions that may be called from the fragment. Intuitively, excluded functions that may be called from within the fragment are considered irrelevant to the operations on c. For the *etcd* example in fig. 5.1, the program fragment obtained for the channel created at line 9 contains all the functions defined in the figure, and its entry is one called `testServer`.

Next, a ***side-effect analysis*** is performed, bottom-up in the SCCs. A function is marked as potentially inducing side-effects to a points-to analysis allocation site if the function itself, or any function it may call directly or transitively, might write to that allocation site. This information, derived from the points-to and call graph analysis, is used in the abstract interpretation phase for estimating the potential side-effects of function calls outside the fragment under analysis.

In the *etcd* example, the call to `s.Close` at line 16 is not relevant for exposing the

bug involving `readyc` and the function being called is not included in the fragment for the `readyc` channel. A naive over-approximation of possible side-effects of that function would assume that the `readyc` and `errc` fields may be overwritten by invoking `s.Close`, losing the guarantee that future reads of these fields yield the references to the initial channels (lines 9–10). The side-effect analysis prevents this by marking the invocation of the `Close` method as not overwriting these fields of `s`.

The results of the pre-analysis phase are used during abstract interpretation, described in the next section.

Localized Abstract Interpretation

This section gives an overview of the abstract interpretation phase by first defining the analysis domain and then describing the abstract semantics in the context of analyzing program fragments. The analysis is designed such that it aborts if it encounters certain difficult situations, which helps ensure a good balance between analysis time, precision, and recall when analyzing a given fragment.

Analysis domain. The abstract domain is a complete lattice that models program behavior flow-sensitively. It is defined as

$$\mathscr{A} = \overbrace{(\mathscr{G} \hookrightarrow \underbrace{(N \times F)}_{\text{control locations}})}^{\text{superlocations}} \to \overbrace{(\mathscr{L} \to \mathscr{V})}^{\text{abstract states}}$$

representing bindings from superlocations (defined below) to abstract states. We next explain each of the components, N, F, $\mathscr{G}$, $\mathscr{L}$ and $\mathscr{V}$.

Using the `ssa` package [83] and the call graph from the pre-analysis, we obtain a control flow graph (CFG) for the given program with a set of nodes N and functions F.

The set $\mathscr{G}$ represents abstract threads. We define $\mathscr{G}$ as the set of **`go`** instructions that appear in the program, i.e., $\mathscr{G} \subseteq N \cup \{\varepsilon\}$, where the special element ε represents the main thread. Intuitively, each thread that may appear at run-time is represented abstractly by the **`go`** instruction where it was spawned.

The domain of $\mathscr{A}$ is the set of *superlocations*, where each abstract thread from $\mathscr{G}$ is bound to a *control location* (or is undefined). In a control location $(n, f) \in N \times F$, the CFG node n represents the next instruction to be executed by the corresponding thread. The function f is the one where the thread started execution. For example, for threads created at line 19 in fig. 5.1, f is the `listenAndServe` function defined at lines 29–33. Intuitively, for a superlocation where $g \mapsto (n, f)$, thread $g \in \mathscr{G}$ is currently at node n and terminates when it leaves f. If g does not map to any control location, it means that no corresponding thread exists. Since control locations are derived from control flow nodes, any (n', f) is a successor of (n, f) if n' is a successor of n in the CFG. Additionally the control location (f_{exit}, f) where f_{exit} is the exit CFG node for f, indicating that a thread is at the exit of the function where it started, has a special successor $(\circ, f)$, indicating that the thread has terminated.

$\mathscr{L}$ is the set of abstract stack and heap locations. These are identified syntactically by the variable declaration sites and allocation sites, respectively, and are further

distinguishable by the abstract thread that allocates them (akin to context-sensitive heaps, or heap cloning, in context-sensitive points-to analysis [76]).

$\mathscr{V}$ is the domain of abstract values. It is a product lattice that combines standard analysis domains for each Go type: constant propagation for basic primitive types, points-to sets for reference types (pointers, interfaces, channel locations,[4] closures, references to built-in dynamic data structures), and a map lattice for aggregate data types, e.g., `struct` values. This is extended with the domain of *abstract channels*, which is a product of four sub-domains:

1. The channel status, represented by a four-element lattice consisting of *OPEN*, *CLOSED*, undefined ($\bot$), and unknown ($\top$). This information is relevant for modeling channel communication (closed channels do not block on reading, whereas sending produces an error) and payload data flow (closed channels with empty buffers produce the zero-value for the payload type).

2. The capacity lattice, which is a constant propagation lattice for natural numbers, keeping track of the channel capacity.

3. The current buffer size lattice, as an interval lattice bounded in height by the number encoded in the capacity, if statically decidable, or a one element lattice (unknown) otherwise.

4. The channel payload, which is itself the abstract value lattice. Possible payload values are joined for channels with capacities greater than 1. For the example in fig. 5.2c, this definition allows the precise modeling of the payload of channel `a` as the reference to channel `b`.

While the abstract domain of channel payloads makes the value lattice inductive, the type system of Go ensures that the height of this lattice is always finite for any given Go program: named types may not have cyclical definitions, except via indirection, which is modeled by points-to sets.

Figure 5.3 depicts the execution paths of the example in fig. 5.1 as a graph. The top of each node represents a superlocation,[5] and the bottom is the associated abstract state. For example, at the superlocation $[\varepsilon \mapsto$ `if err != nil`$]_2$, the channel `readyc` has been previously initialized, and its status is *OPEN* (we here focus on the channel status and omit all the other information being represented by the abstract states). At $[\varepsilon \mapsto$ `<-s.Ready()`$, g_1 \mapsto$ `close(s.readyc)`$]_6$, we model the configuration where the main thread waits on `<-s.Ready()`, and the child thread denoted g_1, which is created at line 19, will next close the `readyc` channel. In abstract states of successors of this superlocation, `readyc` is *CLOSED*. The edges in the graph express the successor relation between the superlocations, and the nodes with thick borders constitute those

[4] To account for potential aliasing, channels are treated as a special kind of objects along with other reference types.

[5] We denote a CFG node by its syntax and omit the function component of control locations for brevity. The subscript labels, e.g., $[\ldots]_1$, uniquely identify the superlocations.

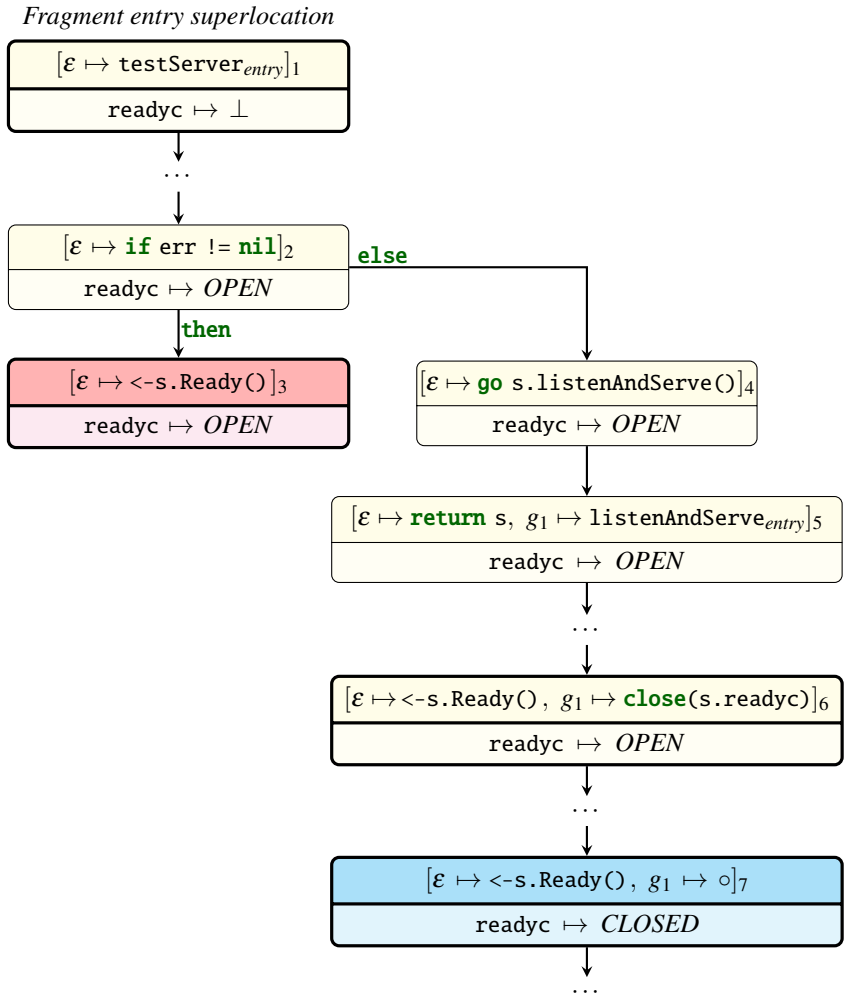


Figure 5.3: Subset of the control flow in the example from fig. 5.1, where superlocations are paired with abstract states.

included in the superlocation graph as explained later. For an execution that takes the 'then' branch, reading from channel `readyc` (at the node marked with red) will block because there is no communication partner. Conversely, on the 'else' branch, g_1 closes `readyc`, unblocking the read operation in the main thread (at the node marked with blue).

Notice that the abstract domain of the analysis has been designed such that it maintains an abstract state for each superlocation rather than tracking state for individual threads. In the example, the status of the `readyc` channel importantly depends on the combination of where the different threads are in the program.

Abstract semantics. Given a program fragment $\mathscr{F}(c)$ with entry f, the abstract interpretation computes an element of the abstract domain $\mathscr{A}$ using a fixpoint computation, as usual in monotone frameworks [35].

Analysis is initiated by assigning an initial abstract state to the entry superlocation $[\varepsilon \mapsto (f_{entry}, f)]$, i.e., the superlocation that only maps the main thread, ε, to the entry of the function f, where f_{entry} is the entry CFG node for f. All other threads are inactive (i.e., their control locations are undefined) at the entry superlocation. Since the fragment entry is generally not a program entry point, the initial abstract state is constructed such that it conservatively models the values of any parameters or free variables using the $\top$ lattice element for the corresponding Go type. To reduce the size of the abstract state, reference types are handled lazily as explained below (see 'Localized analysis').

The abstract interpretation repeatedly applies the transfer functions for the next Go instruction for each active thread, using a worklist algorithm until the least fixpoint is reached. For brevity, we omit a detailed description of the transfer functions; intuitively, they simply model the semantics of Go instructions in the control flow graphs obtained via the `ssa` package [83], according to the abstraction established by the analysis domain. Some interesting analysis design choices are involved, however, in enabling strong updates of control locations, modeling the thread scheduler, and handling calls to code outside the fragment being analyzed, as explained next.

Enabling strong updates of control locations. To provide sufficient analysis precision about channel communication, it is important that each abstract thread represents at most one run-time thread. This property makes it possible to strongly update the CFG node to its successors when processing its transfer function for an abstract thread.[6] If the analysis at a superlocation encounters a **go** instruction that already represents an abstract thread (meaning that a control location is already assigned to that abstract thread), it simply aborts. In complete executions of whole programs, it is very common that a syntactic **go** instruction is encountered multiple times. However, because we analyze not whole programs but relatively small program fragments, the consequences are less severe, as many **go** instructions encountered by the analysis will spawn functions that are not in the analyzed fragment, making the analysis simply ignore them (while treating their potential side-effects conservatively). For example, the fragment of `readyc` in fig. 5.1 may be analyzed independently, even if the fragment entry, `testServer`, may be reached an unbounded amount of times in a complete program execution.

Handling correlated loops. Figure 5.4 shows a common pattern in Go programs. At lines 71–78, a statically unknown number of worker threads are created to compute some values that are then sent to the channel `ch`. At lines 79–81, the results are collected by the main thread. Assume that, in normal executions, the worker threads always take the branch to the send operation at line 75. In this situation, the total number of send operations is equal to the total number of receive operations, so all the threads eventually progress. However, if one of the worker threads does not execute its send operation, the main thread is blocked indefinitely. This program

[6] This notion of strong updates is reminiscent of the one used in points-to analysis [16]. If one abstract thread could represent multiple concrete threads, the CFG node of the control location could only be updated weakly, which would lose the effect of flow sensitivity.

```
71 for i := range list {
72   go func() {
73     ...
74     if ... {
75       ch <- res
76     }
77   }()
78 }
79 for i := range list {
80   ...<-ch
81 }
```

Figure 5.4: Example program where communication involves correlated loops.

exemplifies correlated loops, where two loops perform the same number of iterations as determined by the number of elements in `list`. We take a lightweight approach to handling this pattern by assuming that all loops perform exactly one iteration. This has the advantage of reducing the number of analysis aborts triggered by the conditions discussed above, while allowing the analysis to detect blocking errors as the one in fig. 5.4. If the error in that example is fixed by having the send operations occur unconditionally (e.g., by removing line 74), the analysis reports no error.

Modeling the thread scheduler. Although channels are the recommended mechanism in Go for communicating between threads, the language also supports shared-memory concurrency. The goal of our analysis is not to detect data races but blocking errors caused by channel miscommunication.[7] Nevertheless, the possibility of data races means that a perfectly sound analysis would have to consider all possible interleavings of thread executions, at the level of individual instructions, which quickly leads to a combinatorial explosion in the size of the explored superlocation set.

We alleviate this issue by treating non-communicating instruction sequences as if they were atomic, while preserving the interleavings of channel operations. Intuitively, the abstract interpreter only models thread switches that occur when the currently executing thread is ready to communicate (or has terminated), and communication only occurs when all threads are ready to communicate (or have terminated). Such a coarse-grained modeling of thread scheduling substantially reduces the state space and can only result in missed bugs in the presence of race conditions.

To express this more precisely, we classify each control location as either *communicating* or *silent* to denote whether the operation at its control flow node is a communication operation or not, respectively. Each thread in a superlocation is similarly classified, depending on the type of control location it is bound to. At a *silent superlocation*, at least one thread is silent, and at a *communicating superlocation*, all threads are communicating or terminated.

The analysis computes the least fixpoint $a \in \mathscr{A}$ of the analysis constraints by a series of approximants $\bot = a_0 \sqsubset a_1 \sqsubset a_2 \sqsubset \cdots \sqsubset a_n = a$ using a traditional worklist

[7] This approach is supported by the design philosophy of the Go design team: *"Don't communicate by sharing memory; share memory by communicating"* [81].

algorithm on superlocations [19, 35]. Let ϕ be a superlocation and σ its corresponding abstracting state. Processing ϕ in the worklist algorithm produces a set of transitions, each consisting of a successor superlocation ϕ' and an updated abstract state σ'.

- If ϕ is communicating, we model *inter-processual* data flow. For each enabled communication operation of each thread, the outgoing transitions are computed. Let g be a thread in ϕ where the instruction at the corresponding control location $\theta = \phi(g)$ is an enabled communication operation according to the abstract semantics of the instruction and the abstract state σ. For every ϕ' obtained by modeling the instruction at θ in ϕ, we have that $\theta' = \phi'(g)$ is a successor of θ (and similarly for any g' which is chosen as a potential communication partner, in the case of channel synchronization). Similarly, the corresponding abstract state σ' is obtained by appropriately updating σ according to the abstract semantics of the instruction at θ.

- If ϕ is silent, we model *intra-processual* data flow, by only producing transitions for the next silent thread. The next silent thread is selected by imposing an arbitrary but deterministic order on threads. Let g be the next silent thread of ϕ, and let $\theta = \phi(g)$. The set of outgoing transitions at ϕ is now computed according to the abstract semantics of the (non-communicating) instruction at g relative to the abstract state σ.

In both cases, the next approximant is computed as the least upper bound $a_{i+1} = a_i \sqcup [\phi'_1 \mapsto \sigma'_1] \sqcup \cdots \sqcup [\phi'_k \mapsto \sigma'_k]$ where each pair ϕ'_j, σ'_j for $j = 1, \ldots, k$ is one of the generated transitions.

As an example, since $[\varepsilon \mapsto$ `return s`$, g_1 \mapsto$ `listenAndServe`$_{entry}]_5$ in fig. 5.3 is silent, we apply the intra-processual analysis, first by modeling the sequential operations of ε until `<-s.Ready()` is reached, and then those of g_1 until `close(s.readyc)` is reached (the intermediary steps are elided in the figure). This leads to $[\varepsilon \mapsto$ `<-s.Ready()`$, g_1 \mapsto$ `close(s.readyc)`$]_6$, which is a communicating superlocation, where the analysis models possible progress for each enabled operation of ε and g_1 via inter-processual data flow. As reading from `readyc` is not enabled, due to `readyc` being *OPEN* and ε not having a communication partner, only g_1 can proceed by closing `readyc`. Starting at the successor, we again apply the intra-processual analysis to g_1 (elided), reaching communicating superlocation $[\varepsilon \mapsto$ `<-s.Ready()`$, g_1 \mapsto \circ]_7$. At this point, g_1 is terminated, while ε can proceed by reading from `readyc`, which is now enabled since `readyc` is guaranteed to be closed.

Localized analysis. The fragment being analyzed may contain calls to code outside the fragment. Unknown primitive values (integers, strings, etc.) that originate from such code are modeled conservatively using the $\top$ lattice element for the corresponding type (representing all possible values of that type), as in the construction of the initial abstract state. Code outside the fragment may also affect the abstract state due to side-effects when references escape the fragment. Heap locations that may be affected according to the side-effect analysis (section 5.3) and are of a primitive type are similarly overwritten by $\top$ elements for the corresponding type. For every

```
82 type S struct { ch chan int; val int; flag bool }
83
84 func entry() {
85   s := S{ch: make(chan int, 1), val: 10, flag: false}
86   init(&s)
87   s.ch <- s.val
88 }
89
90 func init(s *S) {
91   if s.flag {
92     s.val = 0
93   }
94 }
```

Figure 5.5: Localized analysis example.

escaping heap location with a reference type, the points-to pre-analysis provides a conservative points-to set that models all possible side-effects to that location.

Unfortunately, Go's standard points-to analysis that we rely on does not support points-to queries to arbitrary heap locations but only to SSA registers. For this reason, our implementation queries the points-to analysis lazily, when the points-to sets of interest reach registers. This also has the effect of reducing the sizes of the points-to sets in the abstract states. However, it causes complications when lazily evaluated points-to sets themselves escape the fragment being analyzed. When that occurs, we pragmatically choose to simply ignore side-effects involving those references. Also, we let the analysis of a fragment abort if a lazily evaluated points-to set contains a reference to a channel in $\mathscr{P}(c)$, as the analysis has likely lost too much precision in that situation.

For the fragment created for the channel allocated on line 85 in the example program in fig. 5.5, the fragment entry is the function `entry` and the function `init` is not included in the fragment. When the analysis reaches the call to `init`, the reference to `s` escapes the fragment. At this point the abstract state for the object stored at that abstract location corresponding to `&s` is $[\texttt{ch} \mapsto \{\texttt{make(chan int, 1)}^{\varepsilon}_{85}\}, \texttt{val} \mapsto 10, \texttt{flag} \mapsto \texttt{false}]$. Here, $\texttt{make(chan int, 1)}^{\varepsilon}_{85}$ denotes a channel allocation on line 85 by the thread ε. The side-effect analysis tells us that the field `val` may be overwritten in the call to `init`, therefore the abstract value for the field `val` is updated to $\top$. We can soundly keep the abstract points-to set for the field `ch` and the constant `false` as the abstract value for the field `flag`, as these fields cannot be written to in `init`. Consequently, the abstract interpreter knows that the only channel that is operated on at line 87 is $\texttt{make(chan int, 1)}^{\varepsilon}_{85}$.

To illustrate the technical issue with channel references in lazily evaluated points-to sets, assume we modify the assignment `s.val = 0` on line 92 such that `s.ch` is instead assigned a newly allocated channel. The abstract value for the `ch` field would then be replaced by a lazily evaluated points-to set when the call to `init` is encountered. After the call, when the channel receive operation is reached on line 87, the lazily evaluated points-to set reaches an SSA register and is expanded by querying

the points-to analysis, which returns the points-to set $\{\texttt{nil}, \texttt{make(chan int, 1)}^{\top}_{85}, \texttt{make(chan int, 1)}^{\top}_{92}\}$, where $\top$ denotes an unknown thread. In this case, the analysis aborts because the expanded points-to set contains the channel the fragment was created for. In our experiments we find that channel values are rarely overwritten, so this situation is not common.

The side-effect analysis is a crucial component that tells the abstract interpreter when it can soundly be assumed that channel values are not overwritten in calls to functions outside the fragment. Despite the technical limitations regarding points-to information and potential for analysis failure, the experimental evaluation (section 5.4) shows that the analysis is able to find many blocking errors.

Detecting Blocking Errors

The abstract interpretation of a given program fragment produces a *superlocation graph*, which is a finite transition system where each node denotes a reachable communicating superlocation or is the initial superlocation of the fragment, and edges are obtained from the transitions described in section 5.3. Given communicating superlocations ϕ_1 and ϕ_2, there is an edge from ϕ_1 to ϕ_2 if the abstract interpreter discovers a sequence of transitions from ϕ_1 to ϕ_2 where all the intermediate superlocations are silent.

The error detection phase is carried out by performing simple model checking on the superlocation graph. Specifically, it checks for every communicating thread g of every superlocation ϕ that there exists at least one path from ϕ to some ϕ' where g has made progress. The absence of such a path indicates a potential blocking error in the fragment.

This approach to detecting blocking errors may have both false positives and false negatives. Since the abstract interpretation phase performs over-approximations, it may discover reachable superlocations that do not correspond to concrete program configurations that are reachable at run-time. Abstract threads in such superlocations may exhibit blocking errors according to the check described above, leading to false positives. Over-approximating whether transitions are enabled can also lead to blocking errors being missed, as the produced transition system may contain spurious paths to superlocations where a thread makes progress.

An error report specifies which thread is potentially blocked and on which line, which channel is involved, and a shortest path in the superlocation graph from the fragment's entry superlocation to the superlocation where the goroutine is blocked. This information aids further diagnosis.

5.4 Evaluation

The proposed approach is implemented in the tool GOAT, on top of existing libraries for Go program analysis, i.e., the Go package loader, parser, type analysis, SSA IR constructor [83], and the points-to and call graph analysis [82]. We evaluate its efficacy by answering the following research questions:

RQ1: What is the precision of the analysis for detecting blocking errors in real-world Go programs?

RQ2: How does the analysis accuracy compare to that of the state-of-the-art Go concurrency bug detection tools GCatch [47] and GFuzz [48]?

RQ3: Does the analysis scale to large code bases?

RQ4: Are the central design choices (treatment of loops and thread scheduling) important for the analysis accuracy and performance?

To answer the research question regarding accuracy and performance, and to compare with GCatch and GFuzz, we use a suite of 6 large real-world Go projects as benchmarks. These are shown in table 5.1. They are mature software systems used in critical production environments. Also, they nearly correspond to the suite of benchmarks used for the evaluation of GFuzz, but we have chosen to exclude Docker as it uses a legacy dependency management and build system. To enable comparison of our results with those of GFuzz, we use the same version of the projects as the GFuzz artifact. We refer to this version of the benchmarks as **suite A**.

The GCatch tool was evaluated on the same projects but in an earlier state, so we manually revert the relevant fixes submitted by the GCatch team such that rediscovering the blocking errors is possible.[8] We refer to this version of the benchmarks as **suite B**.

When we run GOAT on a project, we invoke it once for every package in the project that contains channel creation and consolidate the results over all packages. The pre-analysis phase (section 5.3) is run once for each package, and the abstract interpretation and blocking error detection phases (section 5.3) are run once for each fragment produced by the pre-analysis. We impose a 60 seconds time limit on each run of the abstract interpreter.

The GOAT tool and the experimental data are available at `brics.dk/goat`.

RQ1: Precision

Analysis precision is measured as the ratio between the analysis' reports that are true positives and the total number of reports.

Running GOAT on the 6 real-world Go projects described above results in a set of reports of potential blocking errors. To evaluate the metric the reports must be categorized into true and false positives. Two co-authors manually performed the categorization by inspecting the context of the reported code, and by looking at previously reported and known blocking errors from GCatch and GFuzz, and at the issue trackers for the relevant projects to see if the blocking error had previously been reported or fixed. In cases of doubt due to complex control-flow in the context

[8] Reproducible builds are not supported by older versions of Go. Using the same versions of the projects as GCatch was evaluated on is unfortunately not possible as we could not build them in the old state.

Name	**Description**	**KLOC**	**GitHub stars**
grpc	An implementation of the gRPC remote procedure call system	117	15.5K
etcd	A distributed reliable key-value store	181	39.7K
go-ethereum	An implementation of the Ethereum protocol	368	37.1K
tidb	A distributed HTAP database compatible with MySQL	476	31.2K
prometheus	A monitoring system and time series database	1 186	42.2K
kubernetes	A system for managing containerized applications across multiple hosts	3 453	88.0K

Table 5.1: Go benchmark projects.

	True positives			False positives
Benchmark	GFuzz	Shared	GOAT	GOAT
grpc	7	1	2	6
etcd	3	4	31	11
go-ethereum	34	6	14	27
tidb	7	0	0	0
prometheus	8	3	5	6
kubernetes	15	2	31	30
Total	74	16	83	80

Table 5.2: Bug reports for suite A.

	True positives			False positives	
Benchmark	GCatch	Shared	GOAT	GOAT	GCatch
grpc	2	4	2	6	1
etcd	2	28	39	11	7
go-ethereum	3	9	25	29	15
tidb	5	0	0	0	3
prometheus	1	7	10	6	2
kubernetes	3	5	28	30	7
Total	16	53	104	82	35

Table 5.3: Bug reports for suite B.

of the reported blocking error, or insufficient understanding of a project's code, we conservatively categorized the report as a false positive. None of the bugs we found have yet been reported to the developers (except for the *etcd* bug described in section 5.2).

The results of the categorization are presented in tables 5.2 and 5.3. In total, GOAT reports 99 true positives for suite A (obtained by adding the 'Shared' true positives to those listed in the GOAT column in table 5.2) and 80 false positives, and 157 true positives and 82 false positives are reported when GOAT is run on suite B. GOAT achieves a precision of $99/179 \approx 55\%$ on suite A and a precision of $157/239 \approx 65\%$ on suite B, which is on par with the precision of GCatch. This means that roughly one in every two reports is a true positive, which we consider acceptable for practical use.

On the suite of 6 large real-world Go projects, GOAT *achieves an acceptable true positive ratio of more than* 50%.

RQ2: Comparison with GFuzz and GCatch

We compare the effectiveness of our proof-of-concept blocking error detection tool with two state-of-the-art tools GFuzz and GCatch.

Comparison with GFuzz: For this comparison we run GOAT on the benchmarks in suite A and collect the produced blocking error reports. To obtain the blocking errors reported by GFuzz we do not attempt to run the tool itself, as it relies on a highly non-deterministic fuzzing technique. Instead we use the list of blocking error reports in the GFuzz artifact [46]. The results are summarized in table 5.2.

The true positive reports are separated into three non-overlapping groups: 'Shared' reports are reported by both tools, whereas the GOAT and GFuzz groups denote reports that are produced exclusively by the corresponding tool.

We see that the tools report a nearly disjoint set of blocking errors. This is expected, as the techniques involved are very different. GFuzz only attempts to over-approximate which select branches are chosen in an execution, while GOAT also explores data-dependent control flow, scheduling, and choice of communication partners.

Since GFuzz detects bugs in fuzzed concrete executions, the tools reports few false positives. Nonetheless, imprecision in how GFuzz tracks which goroutines can send on which channels causes it to report 14 false positives according to its authors. The GFuzz artifact additionally contains bugs reported by GCatch when run on suite A. When the true positive reports are combined with the GFuzz reports, GOAT finds 76 bugs that are missed by both GFuzz and GCatch.

Comparison with GCatch: We run both GCatch and GOAT on the benchmarks in suite B and collect the produced blocking error reports. The results are summarized in table 5.3. The true positive reports are again separated into three non-overlapping groups. The false positive reports are not separated in this fashion.

We find that GOAT reports 104 true positives that GCatch misses, whereas GOAT misses 16 true positives that GCatch reports. We inspected the 16 reports that GOAT misses to understand why they are missed. Three reports involve unsupported features, like reflection, `panic` and `recover`, or standard library functions we have not modeled. Three errors are missed due to imprecision in abstract channel properties, resulting from branches in sequential control flow. GCatch handles such situations with more precision by generating different paths for each branching point. Another two errors require precise loop unrolling, handled by GCatch as part of its path unrolling mechanism. One blocking error is missed due to GOAT unsoundly pruning the call graph. The remaining errors are undetected due to aborting the analysis, with the unbounded spawning of threads as the predominant factor.

> GOAT *is able to detect many blocking errors that are missed by the state-of-the-art tools GCatch and GFuzz.*

RQ3: Scalability

For this research question we wish to evaluate whether the program analysis scales to large code bases. We do this by measuring the running time of the analysis phases on suite B (the results for suite A are essentially the same). The results of the measurements are presented in table 5.4. The "Time" column displays the average time per run, while the "95%" column displays the time t such that 95% of the runs complete in less than t seconds.

Across all benchmarks, pre-analysis is performed 476 times, whereas the abstract interpretation phase and the blocking error detection phase are run 11 199 times. On average, 14 minutes are spent in each run of the pre-analysis phase and a second is spent in each run of the abstract interpretation phase. In these experiments we repeat the pre-analysis for all program packages to best measure the potential of our approach in detecting blocking errors. It is also possible to run the pre-analysis once for all packages combined, at the cost of a modest reduction in analysis precision. The largest benchmark, *kubernetes*, is an outlier for the pre-analysis time but is processed quickly by the main analysis phases. *tidb* is an outlier for analysis time where more than 5% of analysis runs for this benchmark time out.

A detailed breakdown of the running time of the abstract interpretation phase is presented in fig. 5.6. Here we see that the vast majority of runs of the abstract interpreter finish within 5 seconds, and that only a small number of runs (70 of 11 199 $\approx 0.6\%$) are aborted due to reaching the time limit. The time required for the blocking error detection phase is negligible.

> *The abstract interpretation and blocking error detection phases of the approach finish quickly in the majority of cases. The total analysis time is dominated by the points-to analysis performed in the pre-analysis phase.*

	Pre-analysis			Main analysis		
Benchmark	Runs	Time	95%	Runs	Time	95%
grpc	96	24 s	46 s	3 023	1.64 s	5 s
etcd	45	42 s	65 s	1 769	0.12 s	1 s
go-ethereum	65	31 s	54 s	3 796	0.55 s	2 s
tidb	14	1 182 s	1 958 s	342	10.99 s	60 s
prometheus	24	67 s	494 s	742	0.22 s	1 s
kubernetes	232	1 651 s	8 924 s	1 527	0.15 s	1 s
Total	476	856 s	7 774 s	11 199	1.02 s	4 s

Table 5.4: Number of runs and running times for pre-analysis and main analysis.

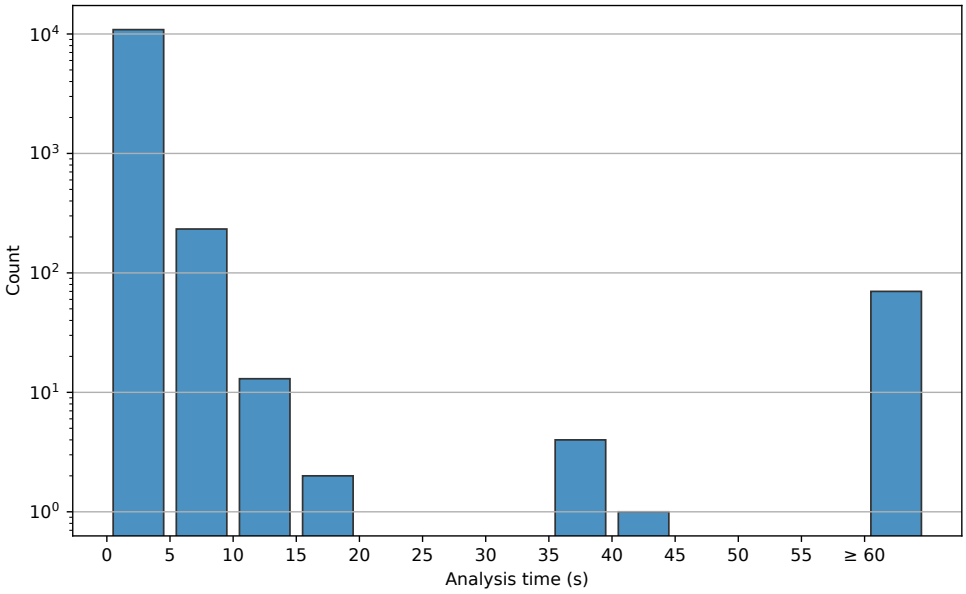


Figure 5.6: Distribution of running times.

Mode	TP	FP	Aborts	Timeouts
Normal	123	53	76%	69
Sound loops	109	46	79%	66
Fine-grained scheduler	113	47	73%	780

Table 5.5: Importance of design choices.

RQ4: Importance of Design Choices

For this research question we wish to evaluate the choice of modeling only one iteration of each loop and the coarse-grained thread scheduler, both explained in section 5.3. We do this by analyzing the benchmarks in suite B with GOAT in three different modes. 'Normal' is the default mode, 'Sound loops' models loops soundly as in normal abstract interpretation, and 'Fine-grained scheduler' replaces the coarse-grained scheduler with a fine-grained scheduler that considers the interleavings of all operations. In table 5.5, we compare each mode by measuring reported true and false positives, timeouts, and the abort ratio for all benchmarks except *kubernetes* (which we exclude in this experiment due to prohibitive pre-analysis times).

Disabling the special treatment of loops increases the number of aborted analysis runs, as expected. The analysis also detects fewer blocking errors: 16 errors are missed compared to a normal run, but only 2 new errors are discovered. The number of false positives also decreases, but at a smaller rate compared to that of true positives.

Modeling all interleavings, including those for silent superlocations, causes a significant increase in the size of the reachable superlocation space explored by the worklist algorithm. For example, analyzing the fragment in fig. 5.1 with the approach from section 5.3 only computes 637 unique superlocations, while 29 629 superlocations are reached if considering all possible interleavings. More generally, the average time for one abstract interpretation run increases from 1.02 s to 9.43 s, the number of timeouts increases substantially, and the number of detected errors decreases without any significant reduction in false positives.

The pragmatic design choices positively impact the analysis by improving the error detection capability and efficiency.

Discussion

Among the threats to the validity of the claims is the choice of benchmarks, which may not be representative of typical Go programs. We have selected benchmarks that are used in prior work to enable comparison, and they are large and high ranked on GitHub. Also, mistakes during the manual inspection of reports may lead to incorrect classification of true positives and false positives; as mentioned we have attempted to conservatively classify difficult cases as false positives.

In our experiments the abstract interpretation phase aborts in 73% of all runs according to the conditions described in section 5.3 and reports no blocking errors

in those cases. Although the tool still finds many blocking errors and has a good ratio between true and false positives according to the experiments, many errors may remain undetected. As an interesting opportunity for future work, it may be possible to increase the completion rate by refining the analysis domain and the heuristics of the pre-analysis.

The GoBench [92] test suite contains 103 *bug kernels*, i.e., Go programs that have been manually synthesized by extracting the critical parts of surrounding code necessary to trigger blocking errors found in real-world Go projects. Of these 103 kernels, 24 are synthesized from blocking errors involving channels and are therefore in the scope of GOAT. We find that GOAT detects 12 of the 24 blocking errors (whereas GCatch detects 10), again indicating that there may be many more errors to be found (provided that the bug kernels are representative of real-world Go programs). Alongside over-approximations and aborting the analysis, another prevalent cause of missing blocking errors is the construction of $\mathscr{P}$ and $\mathscr{F}$, which both GOAT and GCatch rely on. The current heuristics largely group channels intra-procedurally, but several bug kernels expose blocking errors caused by intricate inter-procedural interactions. More extensions for identifying specific inter-procedural patterns (such as C3 in section 5.3) might improve error detection capabilities while keeping the resulting fragments small, which remains to be explored in future work.

Many false positives are due to channel operations that are unreachable in concrete executions but reached by the analysis due to over-approximation. Typical causes of such over-approximations are data-dependent control-flow, spurious cycles in the call graph, imprecision due to channel inclusion in dynamic data structures, or excluding critical channels from the relevant set of a given fragment. Under-approximations also cause false positives. Call graph pruning at highly imprecise call sites may remove edges to functions that should be included in the fragment, and the special treatment of loops and the coarse-grained modeling of the thread scheduler might lead to missing control flow paths. As pointed out in section 5.3, over-approximation in the pre-analysis and abstract interpretation phases may also cause errors to be missed in the blocking error detection phase.

The results of the experiments for RQ3 indicate that GOAT would benefit from more performant analyses used in the pre-analysis phase. Our approach does not require the full points-to solution that the analysis produces, it only needs points-to information for channel operations and for reference values that come from outside the fragment being analyzed. This suggests that a demand-driven points-to analysis would be a good fit for GOAT.

To limit the scope of GOAT, we focus on blocking errors, but the underlying technique may be extended to check for other channel-related properties. The produced abstract states may be used to check for potential safety violations, such as writing to or closing already closed or `nil`-valued channels. The scope of blocking errors can also be extended to include other common Go concurrency primitives, like `Mutex`, `WaitGroup`, and `Cond`.

5.5 Related Work

As stated in section 5.1, GCatch [47] and GFuzz [48] represent the current state-of-the-art for automated detection of blocking errors in real-world Go programs. The key difference to our approach is that GCatch relies on SMT encoding of the execution paths in the program fragments whereas GOAT builds abstract state spaces for the program fragments using abstract interpretation. We believe abstract interpretation provides a natural approach to model the various language features of Go, and that it is flexible for exploring variations of the abstract domain. GFuzz is a dynamic analysis tool that instruments `select` statements to force execution of specific branches and produce alternative case selections via fuzzing. This leads the modified executions to reach execution paths that may be difficult to produce in traditional testing. It also instruments the run-time of Go by collecting relationships between threads and channels, and periodically scans the memory for threads blocked on channels with no future communication partner. As a dynamic analysis tool, GFuzz has high precision for the paths it explores but is restricted to concrete executions.

Combining behavioral types and model checking is another popular approach. Techniques include deadlock detection for programs with synchronous channels by global graph synthesis [59] and session type inference for fenced programs [43], abstracted to a symbolic state space for which safety and k-liveness properties are verified. The Godel checker [44] infers session types and verifies safety and liveness properties defined as μ-calculus encodings via off-the-shelf LTL model checkers. Key limitations to these approaches are difficulties in combining session types and more precise data abstractions, and scalability to real-world Go programs due to lack of coverage of other Go features, e.g., higher-order functions, aliasing, and interfaces. The most recent approach in this family is Gomela [25, 26], which translates Go programs to Promela and uses SPIN [32] for model checking. The experimental results reported for Gomela show good results on small programs but also that it finds relatively few concurrency errors in large, real-world Go programs.

Abstract interpretation has a solid mathematical foundation [19] and has been studied and applied extensively for decades, mostly for single-threaded programs but also for concurrency (see, e.g., [41, 53–55, 87]). Most of the existing techniques are based on thread-modular analysis, without localization to program fragments, and are designed for shared-memory concurrency not involving channels. Previous work on abstract interpretation for channel-based concurrency introduced a notion of process-local static analysis where each abstract thread flow-sensitively models an over-approximation of possible futures as lattice-valued regular expressions [51]. However, it only models communication for a fixed number of threads and synchronous channels, and has only been evaluated on small programs, unlike our approach.

Many of these existing tools detect not only blocking errors but also other kinds of concurrency errors. In principle, GOAT can easily be extended to also scan for safety errors, based on the superlocation graphs it already produces, but we leave that for future work.

GCatch, Gomela and GOAT all achieve scalability by analyzing program fragments

individually. The technique GOAT uses for approximating the behavior of program code outside the fragment being analyzed can be seen as a variation of the "worst-case separate analysis" approach by Cousot and Cousot [21], except that we do not need to compose modular analysis results. Also, we leverage the pre-analysis and we choose to ignore certain potential side-effects involving references as discussed in section 5.3.

5.6 Conclusion

We have shown that localized abstract interpretation is a promising approach to detect blocking errors in programs that use channel-based thread communication. This approach offers an alternative to existing techniques that rely on SMT encoding or bounded model checking of program fragments. The pragmatic design choices make the approach neither sound nor complete, but enable scalability to large code bases and detection of many bugs in practice. The implementation of the approach, GOAT, can detect blocking errors in real-world Go programs that other tools miss, with more than 50% of the reported issues being true positives.

Our experiments also suggest opportunities for further improvements. Provided that the existing collection of small benchmark programs by Yuan et al. [92] is representative of real-world usage of Go, many blocking errors remain beyond reach of existing automated techniques despite the progress obtained by GOAT. It may also be worthwhile to develop more specialized pre-analyses. Furthermore, it may be interesting to extend the analysis to also report safety errors and to model other concurrency primitives, by building on the superlocation graphs produced by GOAT and making further use of the flexibility of abstract interpretation.

5.7 *Epilogue:* Modeling of more Concurrency Primitives

We mentioned an opportunity for future work in the previous section, which is to extend the analysis with support for additional shared memory concurrency primitives available in Go's `sync` package: `Mutex` (locks), `RWMutex` (reader/writer mutual exclusion locks), `WaitGroup`, and `Cond` (condition variables). We will refer to these primitives as *traditional* concurrency primitives. The idea behind this extension is two-fold. One aspect is to give evidence to the flexibility of our approach. The extension should only require simple changes in the pre-analysis phase, changes to the analysis domain to support special abstract representations of the new primitives, and the implementation of transfer functions for Go function calls that manipulate the new primitives. We claim that the bug detection phase, which inspects superlocation graphs produced by the abstract interpretation phase, does not require changes at all. The second aspect is to further characterize the false negative rate of our technique. The main tool that we use to measure false negatives is the GoBench suite [92], mentioned briefly in section 5.4. The study of false negatives requires a set of programs with known bugs, such that when we run our analysis on these programs, we can detect whether we fail to report the known bugs. The experiment we performed to measure

```
 95 import "sync"
 96
 97 type SynchronousCounter struct {
 98   sync.Mutex
 99   cnt int
100 }
101
102 func (sc *SynchronousCounter) inc() {
103   sc.Lock()
104   defer sc.Unlock()
105   sc.cnt++
106 }
107
108 func main() {
109   ch := make(chan int)
110   mu := &sync.Mutex{}
111   counter := &SynchronousCounter{}
112   /* ... */
113 }
```

Figure 5.7: Go program with channel and mutex allocation sites.

false negatives was conducted on a small set of programs, as only 24 of the 103 buggy programs in the GoBench suite contain bugs that are directly (and only) related to channels. By expanding the scope of GOAT to traditional concurrency primitives, we increase the number of relevant programs in the GoBench suite to 68.[9] Provided that the programs are representative of real-world Go programs, increasing the size of the benchmark suite can give us more confidence that the measured false negative rate is also representative. These additional programs contain blocking errors that are caused by misuse of traditional primitives only, but also programs where interplay between traditional and channel concurrency primitives results in bugs. We do not distinguish between these cases in the remainder of the section, but we note that modeling traditional primitives can lead to more blocking errors being reported for channels.

Implementation In the pre-analysis phase we identify creation sites for primitives, analyze how they are used throughout the program, and identify a fragment based on this information. Previously we did this only for channels, but to support traditional primitives, we extend this phase such that it also creates fragments for these. Figure 5.7 contains a small program that allocates a channel, a mutex, and an instance of the custom type `SynchronousCounter`. The program illustrates a common Go idiom where a mutex is embedded inside a struct type, which serves the purpose of protecting against concurrent accesses to the struct's fields, in this case only the integer field `cnt`. The channel allocation is easy to identify syntactically due to the use of the builtin

[9]The remaining programs contain non-blocking bugs, such as data races, which are not in the scope of our technique.

function `make` on line 109. The other concurrency primitives are implemented as regular structs, and can be allocated in various ways. Line 110 contains an example of a direct allocation of a mutex. In the SSA representation of the program, this line is compiled to an allocation instruction for an object of type `sync.Mutex`. Finally, line 111 allocates an instance of `SynchronousCounter`, which implicitly contains a mutex. The SSA allocation instruction for this line is thus the allocation site of both the `SynchronousCounter` object and the embedded mutex. In the SSA translation of the `inc` function, the call to `sc.Lock()` is lowered to a series of instructions that first computes the address of the mutex given the address of the receiver object and then calls the `Lock` method on the mutex object. By querying the pointer analysis for the allocation site of the receiver of the call to `Lock`, we will receive the allocation site of the object the mutex is contained within. It is a slight technicality, but a structure can freely contain multiple fields of the same type, i.e. mutexes, and in this case we would treat all of them as having the same allocation site. The pointer analysis used to connect uses of primitives with their allocation sites does give enough information to discern different primitives embedded in the same structure, but we chose to keep the implementation simple and aligned with our existing handling of channels.

Uses of channels come in the form of `select` statements and send-, receive- and close operations. However, we identify uses of traditional primitives by looking for calls to special methods, such as `Lock` and `Unlock` on lines 103 and 104. This generally requires pointer analysis information, as calls to these methods can happen via dynamic dispatch through interface values. Extra care must be taken for condition variables, which wrap a reference to a lock. A call to `Wait` on a condition variable is not relevant to the object only, but also to the wrapped lock, which will be unlocked and locked during the execution of `Wait`. Condition variables are also a little bit special in that they are typically allocated with the constructor function `sync.NewCond`, as opposed to being allocated directly as seen in fig. 5.7. Presumably, this is to prevent users from forgetting to supply a reference to the mutex which should be wrapped by the condition variable. This is slightly problematic for us, as any condition variable allocated with the constructor will have the same syntactic allocation site due to context insensitivity in the pointer analysis result. To alleviate this issue, we pre-process the analyzed code and automatically inline calls to this function.

Otherwise, the rest of the pre-analysis phase proceeds similarly as described in section 5.3. $\mathscr{P}(c)$ is computed for traditional primitives as well as channel primitives, and they can contain both types of primitives at the same time. Note that the only relevant criterion for inclusion of a traditional primitive in $\mathscr{P}(c)$ is **C1**, as the others are specific to channel primitives.

For the main analysis phase, we must extend the domain of abstract values $\mathscr{V}$ with sub-domains for the new concurrency primitives. For values of type `sync.Mutex` we use a simple lattice for a boolean flag, identical to the one used for the status of channels. Instead of *OPEN* and *CLOSED* we use *UNLOCKED* and *LOCKED*. For reader/writer mutual exclusion locks (`sync.RWMutex`) we use a product lattice between the simple mutex lattice for the exclusive lock and a constant propagation lattice on non-negative integers that models the number of held read-locks. The

`sync.WaitGroup` primitive is a wrapper around a thread-safe counter, where threads can choose to suspend execution until the counter reaches 0. We abstractly model this with a constant propagation lattice for non-negative integers (a run-time error occurs if the counter becomes negative). We model the final kind of primitive, condition variables, with elements from the points-to lattice. This element shall contain the over-approximated set of mutexes that the condition variable may wrap at run-time.

The implementation of transfer functions for instructions that manipulate the traditional concurrency primitives – which are now classified as communicating – follows naturally from the definitions of their abstract domains. For instance, this is a constraint-based formulation of the transfer function for the lock operation on mutexes:

$$
\begin{aligned}
&\phi_i(g) = \texttt{mu.Lock()} \\
&o \in \sigma_i(\texttt{mu}) \setminus \{\texttt{nil}\} \\
&\mathit{UNLOCKED} \sqsubseteq \sigma_i(o) \\
&\Rightarrow \\
&o \in \sigma_j(\texttt{mu}) \\
&\mathit{LOCKED} \sqsubseteq \sigma_j(o)
\end{aligned}
$$

Here σ_i is the abstract state associated with superlocation ϕ_i, and σ_j is the abstract state associated with the successor superlocation $\phi_j = \phi_i[g \mapsto \text{succ}(\texttt{mu.Lock()})]$. It says that, if there is a thread g in superlocation ϕ_i that is ready to perform a lock operation, if the variable `mu` corresponding to the operated-on mutex contains a non-nil reference to a mutex, *and* if the referenced mutex may be unlocked, then the variable can still point to the referenced mutex in the successor superlocation, and the abstract value for the mutex is at least *LOCKED*.

The semantics of waiting on a condition variable is complex, so we split a call to `cv.Wait()` into three synthetic instructions (control locations). The first instruction is equivalent to unlocking the mutex wrapped in the condition variable. After this instruction is executed, the thread progresses to a *waiting* state where it can be woken by another thread calling `cv.Signal()` on the same condition variable. If this happens, the waiting thread transitions to the final synthetic instruction, which is equivalent to locking the mutex wrapped in the condition variable.

Experiments With the implementation in place we can move on to the experiments. The first experiment is to run our extended tool on the 68 relevant buggy programs in the GoBench suite to determine if our approach works at all for traditional concurrency primitives. GOAT is able to detect bugs in 38 of these programs, which gives a false negative rate that is slightly better than what we experienced for the previous restricted set of programs (recall that GOAT detected bugs in 12 of 24 programs). Common reasons for missed bugs are explained in section 5.4, and they also apply to bugs involving traditional concurrency primitives.

We have then repeated the experiment on the 6 large real-world Go programs to determine whether we can find bugs involving traditional concurrency primitives in

those, and whether our approach still scales to large programs with our extensions. Because the difference between suite A and suite B is related to channels only, we chose to run the experiments on suite A for simplicity.

Across all benchmarks, the abstract interpretation and blocking error detection phases are run 24 803 times. This means that approximately twice as many fragments are formed when we analyze both traditional and channel concurrency primitives. Analysis still completes within a second in most cases, and the analysis success rate is similar to what we presented in section 5.4.

With respect to bug detection, we saw in table 5.2 that 170 potential blocking errors were reported when the analysis was restricted to channel primitives only. With our extensions, the analysis now reports 246 potential blocking errors in total, and 76 of them are for operations on traditional concurrency primitives. Unfortunately, none of these reports appear to be true positives. We looked into the circumstances around the false positives and can report some findings categorized by the type of primitive that the analysis thinks an operation may block indefinitely on.

Typical uses of condition variables associate a condition, i.e. some predicate on the program's state, with each variable. If the condition is not satisfied, the condition variable can be *waited* on until relevant parts of the program's state changes, which is *signaled* by another thread. Proper use of a condition variable (typically) requires the variable to be waited on in a loop, like so:

```
while (condition is not satisfied) { cv.Wait() }
```

The abstract state inferred by GOAT before such loops is typically not precise enough to determine (yes/no) if the condition is satisfied, so for soundness both possibilities are modeled. This causes the analysis to model that `cv.Wait()` can be called in scenarios where it cannot, and where there are no threads that will signal the waiting thread in the future. This is the predominant reason for false positives involving condition variables. This issue can potentially be overcome with standard abstract interpretation techniques for precise reasoning about specific pieces of program state, such as relational analysis [52, 54, 55] and path-sensitivity [10, 31], and is an opportunity for future work.

For false positives involving WaitGroup primitives, we note that wait groups are typically used to wait for a number of concurrent child threads to finish their assigned work. In such cases, a wait group is initialized with the number of child threads that will be spawned, after which the threads are spawned — often in a loop — and execution in the parent thread is suspended until the counter in the wait group reaches 0. When a child thread terminates it decrements the counter in the wait group. In many cases the value that the wait group is initialized with is not statically determinable because it is computed as the run-time size of some data structure. In such cases, the analysis will never report a potential blocking error, as it cannot eliminate the possibility of the wait group counter being 0 when the parent thread suspends execution. The use of a slightly more complex abstract domain for wait groups, that has an additional lattice element modeling strictly positive integers, would allow for some reasoning about the presence of potential blocking errors in these

situations. But there are also cases where the counter is initialized with a value that *is* statically determinable, and an unfortunate interaction with our special handling of loops directly causes false positive reports. When we fix the number of loop iterations for a loop that is supposed to decrement a counter N times, we also change the semantics such that only a single decrement is modeled by the abstract interpreter. This causes the modeled counter in the wait group to never reach 0, triggering a potential error to be reported, even though the program may use wait groups correctly. We can create a filter for potential blocking errors that can detect when the report is a likely false positive due to this interaction.

Surprisingly, there are no reports involving operations on locks. It seems that the status of abstractly modeled locks often becomes unknown, which prevents bugs from being reported by our bug detector. A common bug pattern involving locks is to forget to unlock a lock along *some* execution paths in a function, for instance if some exceptional condition occurs. In such a scenario the abstractly modeled status of the lock would become the join of the status along all execution paths, i.e. unknown. GCatch [47] finds 29 bugs involving locks in the analyzed benchmarks, specifically by looking for common patterns of misuse of locks, such as the one mentioned above, so there are clearly some bugs that our tool may be able to detect. However, our approach does not attempt to identify such patterns, but instead looks for program configurations where a thread is blocked indefinitely. Forgetting to unlock a lock will only cause a blocking error if a thread later attempts to acquire the same lock. Standard abstract interpretation extensions to our technique, such as path-sensitivity, may be able to increase the analysis' ability to reason precisely about the status of locks, which can lead to more bugs being reported. Such extensions are an opportunity for future work.

In conclusion, while we were able to find known blocking errors involving traditional concurrency primitives in the GoBench suite, the results from experiments in this section are mostly negative. A common theme is that imprecision in the analysis prevents bugs from being reported, which applies to channel concurrency primitives as well. In the next section we will explore how GOAT can detect bugs, even when faced with imprecision in the main analysis.

5.8 *Epilogue:* Sound Blocking Error Detection

Static analyses aimed at sound bug detection usually work by first over-approximating the behavior of the analyzed program. Then, if the over-approximated model of the program contains a buggy execution, a report is issued. Designing a bug detector in this way makes it easy to argue for its soundness, as any behavior that can be exhibited in a concrete execution is guaranteed to be captured by the over-approximation. The bug detector outlined in section 5.3 is not designed in this way. As described earlier, it is unsound and is prone to false negatives, i.e. it can fail to detect bugs that are present in the analyzed program.

Recall that our approach consists of multiple phases: the abstract interpretation phase produces a program model in the form of a superlocation graph, and the graph is inspected in the bug detection phase where potential blocking errors are reported. From program analysis theory we know how to make abstract interpreters sound, however, pragmatic design choices (deterministic thread scheduling, fixed loop iterations, ignored side-effects of some instructions, etc.) inherently make our produced models unsound. Therefore, any non-trivial bug detector we design, that identifies potential bugs by inspecting the model produced by the abstract interpretation phase, will not be able to soundly report bugs wrt. the concrete semantics of the analyzed Go program. The interesting question that is explored in this section is whether it is possible to design a blocking error detector that is sound (and possibly complete) wrt. the semantics of superlocation graphs (as opposed to the semantics of the analyzed Go program).

To understand when GOAT can fail to detect a bug, let us look at the bug detection algorithm in a little more detail. If the abstract interpretation phase (section 5.3) finishes successfully, its output is a graph with communicating superlocations ϕ_i as nodes (plus the initial superlocation). Each ϕ_i has an abstract, finite representation of a potentially infinite set of concrete configurations $\gamma(\phi_i)$.[10] The edge relation is over-approximated as follows: If there exists a concrete configuration c_i in $\gamma(\phi_i)$, and it is possible to end up in configuration $c_j \in \gamma(\phi_j)$ (where ϕ_j is communicating) from c_i by executing at most one communicating instruction, followed by a series of silent instructions, there is an edge from ϕ_i to ϕ_j in the superlocation graph. The set of configurations that are reachable from the initial superlocation in this graph ($\bigcup_{\phi_{\text{initial}} \to^* \phi_i} \gamma(\phi_i)$) over-approximates the set of communicating configurations that are reachable in a concrete execution of the program. This is positive from a soundness perspective, as we want to check that, for each non-terminated thread in each reachable communicating configuration, the thread can make progress in some finite number of steps. By checking this property for a set of configurations that is known to be larger than the set of configurations reachable in a concrete execution, we will not miss any bugs present in the analyzed program. However, the property that is checked by our current bug detector is a different one. Namely it checks that, for each abstract thread g in each reachable superlocation ϕ_i, there is a path to another superlocation ϕ_j where the thread has made progress. Consider a scenario where the path consists only of a single edge $\phi_i \to \phi_j$. From the definition of the transition relation, we know that there exists a concrete configuration c_i in $\gamma(\phi_i)$ such that c_j is in $\gamma(\phi_j)$ and g makes progress when executing instructions from c_i to c_j. There may be another $c_i' \neq c_i$ in $\gamma(\phi_i)$ where some of the instructions that are executed between c_i and c_j are disabled, and where there does not exist a path to a configuration where the thread makes progress at all.

For a concrete example of such a case, let us inspect the program in fig. 5.8 and its corresponding superlocation graph. The program only has a single thread, the main thread, which performs some operations on an asynchronous channel initialized with

[10] Assume that the concretization function γ concretizes both the superlocation and its associated abstract state.

```
115 func main() {
116   ch := make(chan int, 1)
117
118   if ... {
119     ch <- 10
120   }
121
122   <-ch
123 }
```

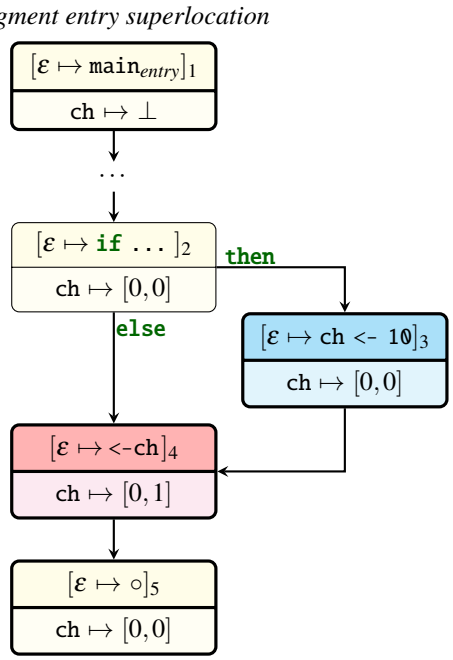


Figure 5.8: Go program with a potential blocking error and the superlocation graph for the only fragment in the program.

a capacity of 1. The value of the branch condition in the if-statement is statically unknown. If the then branch is taken, a message is put into the channel. At this point the analysis knows that the channel is empty, which is indicated by the buffer interval $[0,0]$, so this operation is definitely enabled.[11] After the send operation, the abstract value for the buffer of ch is $[1,1]$. The control flow is then merged after the if-statement, which results in a merge of the abstract values for the channel's buffer. If the then branch is taken, the buffer is $[1,1]$, otherwise it is $[0,0]$. The result of the merge $[1,1] \sqcup [0,0]$ is $[0,1]$. This is depicted in the abstract state associated with the superlocation node $[\varepsilon \mapsto \texttt{<-ch}]_4$. Concretization of the channel at this point results in two concrete channels: one that is empty and one that is full. The next operation to be performed is a channel receive operation. This operation is enabled when the channel is nonempty. Since the abstract configuration has a concretization where the channel is nonempty, the superlocation graph models the possibility of the operation succeeding with an edge to the superlocation $[\varepsilon \mapsto \circ]_5$ where the main thread has terminated.

With the superlocation graph in place, the bug detection algorithm proceeds to check for potential blocking errors in each reachable communicating superlocation. In both of the interesting superlocations, $[\varepsilon \mapsto \texttt{ch <- 10}]_3$ and $[\varepsilon \mapsto \texttt{<-ch}]_4$, there is a path in the graph to a superlocation where the main thread makes progress, which

[11] Only the lattice sub-element abstracting the channel's buffer size is included in the figure, as the other sub-elements are irrelevant in this example.

means that no bugs are reported. However, the program does contain a potential blocking error! When the then branch is *not* taken, the channel's buffer will be empty when the main thread encounters the receive operation (`<-ch`), and it will therefore be blocked indefinitely. This is a false negative. The absence of a path where a communicating thread makes progress does indicate a blocking error, but it is not a necessary condition. It under-approximates the presence of blocking errors. The underlying problem is that the edges in the superlocation graph represent transitions that *may* be enabled.

While the program in fig. 5.8 is unrealistic, scenarios that lead to imprecise abstract states in a similar way are common in practice. They arise whenever abstract channels manipulated by the abstract interpreter concretize to more than one concrete channel, which typically happens because a thread performs a communicating operation conditionally.

One way to make the bug detector sound is to use a different, under-approximating, edge relation when checking for the presence of unblocking paths. Formally, we define the new "*must*" edge relation such that ϕ_i and ϕ_j are connected only if not some but *all* c_i in $\gamma(\phi_i)$ can end up in (possibly different) $c_j \in \gamma(\phi_j)$ by executing at most one communicating instruction, followed by a series of silent instructions. This edge relation is simple to derive given the original over-approximated relation. Each edge, corresponding to the execution of one communicating instruction, is checked in turn. The edge is discarded if the concretization of the source abstract configuration contains a concrete configuration where the instruction is disabled, which is trivial to decide given the abstract state and type of instruction. Notice that this excludes the edge between superlocations $[\varepsilon \mapsto \texttt{<-ch}]_4$ and $[\varepsilon \mapsto \circ]_5$ in the example program, which leads to a blocking error being reported.

We have re-run the bug detection experiment on the 6 large real-world Go projects, described in section 5.4, with the above sound implementation of the bug detector. For simplicity we restrict the experiment to suite A. Across all 6 benchmark projects, 1 954 bugs are reported. This is significantly higher than the 170 reports issued by the original bug detector. Due to the high number of bug reports, it is not feasible to classify all of them as true or false positives by manual inspection, which is what we did for the original 170 reports. Instead, we randomly sample 50 new reports and classify only those. According to our best-effort and conservative classification, all of these new reports are false positives. Scenarios that lead to imprecise abstract values for channels, such as channels that are allocated multiple times in a fragment, and communication operations that are performed conditionally, will often lead to a bug report with the sound detector. However, our experiments show that such scenarios are poor indicators of actual blocking bugs.

Note that the new "*must*" edge relation used to find unblocking paths is not fully precise. Consider the program and associated superlocation graph in fig. 5.9. The program is similar to the one presented in fig. 5.8, but an additional thread is spawned that attempts to send a message on the provided channel. In the associated abstract state for the superlocation ϕ_5 (corresponding to $[\varepsilon \mapsto \texttt{<-ch},\ g_1 \mapsto \texttt{ch <- 5}]_5$) the abstract value for the buffer of `ch` is $[0, 1]$. For both discovered transitions $\phi_5 \to \phi_6$ (corresponding to

```
124 func main() {
125   ch := make(chan int, 1)
126
127   if ... {
128     ch <- 10
129   }
130
131   go f(ch)
132
133   <-ch
134 }
135
136 func f(ch chan int) {
137   ch <- 5
138 }
```

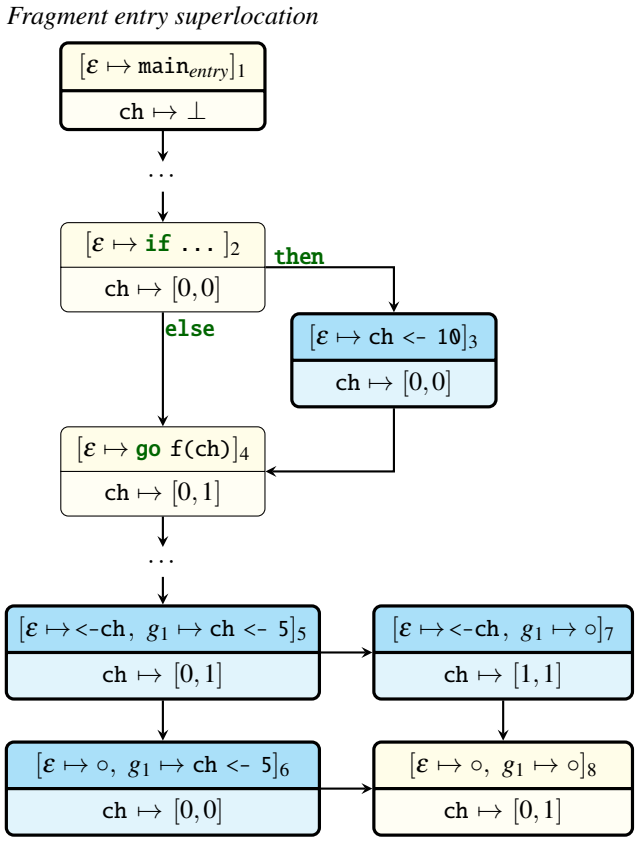


Figure 5.9: Go program with no blocking error and the superlocation graph for the only fragment in the program.

the execution of <-ch by the main thread) and $\phi_5 \to \phi_7$ (corresponding to the execution of ch <- 5 by g_1), there is a concrete configuration in $\gamma(\phi_5)$ where the transition is disabled. Therefore the transitions are *not* included in the derived edge relation, which causes potential blocking errors to be reported for both operations. However, it is not possible for the main thread nor g_1 to become blocked indefinitely in any execution of the program, so these reports are false positives. There is a dependency between the enabledness of the instructions: when one of the transitions is disabled, the other must necessarily be enabled. It is an interesting opportunity for future work to design a bug detector for superlocation graphs that takes these dependencies into account, to limit the sources of imprecision to the abstract interpretation phase only.

Bibliography

[1] Radoslaw Adamus, Tomasz Marek Kowalski, and Jacek Wislicki. A step towards genuine declarative language-integrated queries. In *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Lódz, Poland, September 13-16, 2015*, volume 5, pages 935–946. IEEE, 2015. doi: 10.15439/2015F156. 59

[2] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, volume 952 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 1995. doi: 10.1007/3-540-49538-X_2. 36

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. 18, 60, 70

[4] Matthew Arnold, Stephen J. Fink, Vivek Sarkar, and Peter F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 2000), Boston, MA, USA, January 18, 2000*, pages 52–64. ACM, 2000. doi: 10.1145/351397.351416. 32

[5] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. doi: 10.1109/JPROC.2004.840305. 57

[6] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003. doi: 10.1145/857076.857077. 57

[7] Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. NullAway: Practical Type-Based Null Safety for Java. In Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo, editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 740–750. ACM, 2019. doi: 10.1145/3338906.3338919. 5

[8] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Clash of the lambdas. *CoRR*, abs/1406.6631, 2014. 28, 29, 32

[9] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. Streams a la carte: Extensible pipelines with object algebras. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 591–613. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.591. 34, 51, 58

[10] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 237–251. ACM, 1998. doi: 10.1145/268946.268966. 93

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 243–262. ACM, 2009. doi: 10.1145/1640089.1640108. 36, 42

[12] Zoran Budimlic and Ken Kennedy. Optimizing Java: theory and practice. *Concurrency - Practice and Experience*, 9(6):445–463, 1997. 47, 57

[13] Zoran Budimlic and Ken Kennedy. Static interprocedural optimizations in Java. Technical report, Center for Research on Parallel Computation, Rice University, Technical Report CRPC-TR98746, 1998. 47, 57

[14] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015. doi: 10.1007/978-3-319-17524-9_1. 5

[15] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, pages 152–161. ACM, 1986. doi: 10.1145/12276.13327. 45

[16] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 296–310. ACM, 1990. doi: 10.1145/93542.93585. 18, 36, 44, 76

[17] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM*

SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999, pages 1–19. ACM, 1999. doi: 10.1145/320384.320386. 35, 46, 47, 57

[18] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. Iterative data-flow analysis, revisited. Technical report, 2004. 16

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. 11, 12, 78, 88

[20] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 269–282. ACM Press, 1979. doi: 10.1145/567752.567778. 11

[21] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002. doi: 10.1007/3-540-45937-5_13. 89

[22] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 315–326. ACM, 2007. doi: 10.1145/1291151.1291199. 20

[23] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer, 1995. doi: 10.1007/3-540-49538-X_5. 44

[24] David Detlefs and Ole Agesen. Inlining of virtual methods. In *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 258–278. Springer, 1999. doi: 10.1007/3-540-48743-3_12. 32, 46

[25] Nicolas Dilley and Julien Lange. Bounded verification of message-passing concurrency in Go using Promela and Spin. In *Proceedings of the 12th International Workshop on Programming Language Approaches to Concurrency-*

and Communication-cEntric Software, PLACES@ETAPS 2020, Dublin, Ireland, 26th April 2020, volume 314 of *EPTCS*, pages 34–45, 2020. doi: 10.4204/EPTCS.314.4. 66, 88

[26] Nicolas Dilley and Julien Lange. Automated verification of Go programs via bounded model checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1016–1027. IEEE, 2021. doi: 10.1109/ASE51524.2021.9678571. 66, 88

[27] Julian Dolby and Andrew A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA) '98), Vancouver, British Columbia, Canada, October 18-22, 1998*, pages 1–20. ACM, 1998. doi: 10.1145/286936.286943. 57

[28] Julian Dolby, Stephen J. Fink, and Manu Sridharan. T.J. Watson Libraries for Analysis, 2010. URL `http://wala.sourceforge.net/`. 42, 60

[29] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 57–76. ACM, 2007. doi: 10.1145/1297027.1297033. 52

[30] Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, pages 223–232. ACM, 1993. doi: 10.1145/165180.165214. 20

[31] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. In Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors, *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980*, pages 68–82. ACM Press, 1980. doi: 10.1145/567446.567454. 93

[32] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23 (5):279–295, 1997. doi: 10.1109/32.588521. 88

[33] Manohar Jonnalagedda and Sandro Stucki. Fold-based fusion as a library: a generative programming pearl. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015*, pages 41–50. ACM, 2015. doi: 10.1145/2774975.2774981. 58

[34] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *J. ACM*, 23(1):158–171, 1976. doi: 10.1145/321921.321938. 16

[35] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977. doi: 10.1007/BF00290339. 11, 42, 75, 78

[36] Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. Safe automated refactoring for intelligent parallelization of Java 8 streams. *Science of Computer Programming*, page 102476, 2020. ISSN 0167-6423. doi: 10.1016/j.scico.2020.102476. 58

[37] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. An empirical study on the use and misuse of Java 8 streams. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*, volume 12076 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2020. doi: 10.1007/978-3-030-45234-6_5. 34

[38] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973. doi: 10.1145/512927.512945. 11

[39] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 285–299. ACM, 2017. doi: 10.1145/3093333.3009880. 20, 28, 29, 32, 58

[40] Stephen Cole Kleene. *Introduction to Metamathematics*. Princeton, NJ, USA: North Holland, 1952. 14

[41] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 799–809. ACM, 2016. doi: 10.1145/2950290.2950291. 88

[42] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection: literature review and empirical study. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 507–518. IEEE / ACM, 2017. doi: 10.1109/ICSE.2017.53. 63

[43] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off Go: Liveness and safety for channel-based programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages,*

POPL 2017, Paris, France, January 18-20, 2017, pages 748–761. ACM, 2017. doi: 10.1145/3009837.3009847. 88

[44] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1137–1148. ACM, 2018. doi: 10.1145/3180155.3180157. 88

[45] Ondrej Lhoták and Laurie J. Hendren. Scaling Java points-to analysis using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2003. doi: 10.1007/3-540-36579-6_12. 36, 42, 61

[46] Ziheng Liu, Yu Liang, Shihao Xia, Linhai Song, and Hong Hu. GFuzz ASPLOS 2022 #710 Artifact, December 2021. 83

[47] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in Go software systems. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 616–629. ACM, 2021. doi: 10.1145/3445814.3446756. 66, 70, 71, 81, 88, 94

[48] Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. Who goes first? detecting Go concurrency bugs via message reordering. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 888–902. ACM, 2022. doi: 10.1145/3503222.3507753. 66, 81, 88

[49] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015. doi: 10.1145/2644805. 63

[50] Davood Mazinanian, Ameya Ketkar, Nikolaos Tsantalis, and Danny Dig. Understanding the use of lambda expressions in Java. *PACMPL*, 1(OOPSLA): 85:1–85:31, 2017. doi: 10.1145/3133909. 28

[51] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Process-local static analysis of synchronous processes. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 284–305. Springer, 2018. doi: 10.1007/978-3-319-99725-4_18. 88

[52] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21-23, 2001, Proceedings*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001. doi: 10.1007/3-540-44978-7_10. 93

[53] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Log. Methods Comput. Sci.*, 8(1), 2012. doi: 10.2168/LMCS-8(1:26)2012. 88

[54] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014. doi: 10.1007/978-3-642-54013-4_3. 93

[55] Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, Daniel Kästner, Stephan Wilhelm, and Christian Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, January 2016. URL `https://hal.archives-ouvertes.fr/hal-01271552`. 88, 93

[56] Anders Møller and Oskar Haarklou Veileborg. Eliminating abstraction overhead of java stream pipelines using ahead-of-time program optimization. *Proc. ACM Program. Lang.*, 4(OOPSLA):168:1–168:29, 2020. doi: 10.1145/3428236. 8

[57] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017. doi: 10.1007/s10664-017-9512-6. 34, 56

[58] Derek Gordon Murray, Michael Isard, and Yuan Yu. Steno: automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 121–131. ACM, 2011. doi: 10.1145/1993498.1993513. 58

[59] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent Go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 174–184. ACM, 2016. doi: 10.1145/2892208.2892232. 88

[60] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, pages 43–48. ACM, 2004. doi: 10.1145/996821.996836. 18, 42

[61] Oracle. Jdk 8, 3 2014. URL https://openjdk.java.net/projects/jdk8/. 28

[62] Oracle. java.util.stream documentation for jdk 8, 2014. URL https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html. 28

[63] Oracle. Java microbenchmarking harness, 2014. URL http://openjdk.java.net/projects/code-tools/jmh/. 52

[64] Stack Overflow. 2022 Developer Survey. URL https://survey.stackoverflow.co/2022/. 3

[65] Nick Palladinos and Kostas Rontogiannis. Linqoptimizer: An automatic query optimizer for linq to objects and plinq, 2014. URL http://nessos.github.io/LinqOptimizer/. 29, 58

[66] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, pages 116–127. ACM, 1992. doi: 10.1145/143095.143125. 35

[67] Aleksandar Prokopec and Dmitry Petrashko. Scalablitz: Lightning-fast scala collections framework, 2013. URL https://scala-blitz.github.io/. 29, 58

[68] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, BC, Canada, October 22-23, 2017*, pages 29–40. ACM, 2017. doi: 10.1145/3136000.3136002. 30

[69] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953. 11

[70] John Rose. Hotspot-dev mailing list: Perspectives on streams performance, 3 2015. URL http://mail.openjdk.java.net/pipermail/hotspot-compiler-dev/2015-March/017278.html. 28

[71] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 598–608. IEEE Computer Society, 2015. doi: 10.1109/ICSE.2015.76. 5

[72] Ulrik Pagh Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for Java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003. doi: 10.1145/778559.778561. 58

[73] Denys Shabalin and Martin Odersky. Interflow: interprocedural flow-sensitive type inference and method duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala, SCALA@ICFP 2018, St. Louis, MO, USA, September 28, 2018*, pages 61–71. ACM, 2018. doi: 10.1145/3241653.3241660. 58

[74] Micha Sharir and Amir Pnueli. *Two approaches to interprocedural data flow analysis*, chapter 7, pages 189–234. Prentice-Hall, 1981. 43

[75] Yannis Smaragdakis and George Kastrinis. Defensive points-to analysis: Effective soundness via laziness. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 23:1–23:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi: 10.4230/LIPIcs.ECOOP.2018.23. 63

[76] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 17–30. ACM, 2011. doi: 10.1145/1926385.1926390. 18, 42, 74

[77] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi: 10.4230/LIPIcs.ECOOP.2016.22. 42, 57, 61

[78] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, 2019. doi: 10.1145/3290361. 61

[79] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 387–400. ACM, 2006. doi: 10.1145/1133981.1134027. 36, 42, 60

[80] Bjarne Steensgaard. Points-to analysis in almost linear time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,*

Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996, pages 32–41. ACM Press, 1996. doi: 10.1145/237721.237727. 18

[81] The Go Authors. Share memory by communicating. `https://go.dev/doc/effective_go`, 2010. 77

[82] The Go Authors. Points-To analysis and Call Graph construction for Go, 2022. URL `https://pkg.go.dev/golang.org/x/tools/go/pointer`. 70, 80

[83] The Go Authors. Static Single Assignment Intermediate Representation for Go, 2022. URL `https://pkg.go.dev/golang.org/x/tools/go/ssa`. 73, 76, 80

[84] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 865–878. ACM, 2019. doi: 10.1145/3297858.3304069. 66

[85] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999. URL `https://dl.acm.org/citation.cfm?id=782008`. 61

[86] Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Detecting blocking errors in go programs using localized abstract interpretation. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 32:1–32:12. ACM, 2022. doi: 10.1145/3551349.3561154. 9

[87] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 391–402. ACM, 2016. doi: 10.1145/2970276.2970337. 88

[88] Philip Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile time. In Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr., editors, *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*, pages 45–52. ACM, 1984. doi: 10.1145/800055.802020. 20

[89] Philip Wadler. Listlessness is better than laziness II: composing listless functions. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects, Proceedings of a Workshop, Copenhagen, Denmark, October 17-19, 1985*,

volume 217 of *Lecture Notes in Computer Science*, pages 282–305. Springer, 1985. doi: 10.1007/3-540-16446-4_16. 20

[90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990. doi: 10.1016/0304-3975(90)90147-A. 20, 58

[91] Richard C. Waters. Automatic transformation of series expressions into loops. *ACM Trans. Program. Lang. Syst.*, 13(1):52–98, 1991. doi: 10.1145/114005.102806. 59

[92] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. GoBench: A benchmark suite of real-world Go concurrency bugs. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 187–199. IEEE, 2021. doi: 10.1109/CGO51591.2021.9370317. 87, 89