

Abstract garbage collection is the application of GC to an abstract interpreter. It is known that abstract GC significantly improves both the interpreter’s precision and performance. In this work, we focus on the efficient integration of abstract GC.

To maximise its precision and performance benefits, abstract GC needs to be applied frequently; at the same time, frequent applications bring about their own performance overhead. We propose a novel approach for abstract GC based on reference counting, which we call abstract reference counting. The approach is able to immediately eliminate all garbage, maximising abstract GC's precision and performance benefits. Yet, it does not require frequent heap traversals, therefore adding no significant overhead. To ensure all garbage is eliminated, we augment abstract reference counting with cycle detection, counting references at the level of strongly connected components. The empirical evaluation shows that the resulting approach achieves both the optimal precision and significantly improved performance compared to existing approaches to abstract GC.

Second, we consider the integration of abstract GC for abstract interpreters that do not explicitly model continuations (such as abstract definitional interpreters). These offer extra precision and performance benefits by eliding an over-approximation of the continuation, and instead making use of recursive evaluation rules that inherit perfect stack precision from their defining language. Existing techniques either fail to exploit the full benefit of both abstract GC and such “big-step” abstract interpreters, and/or require complicated modifications to the interpreter. In addition, they all lack scalability, as abstract GC is not compatible with global store widening. We present DSS, a novel yet simple approach to integrating abstract GC into abstract definitional interpreters. DSS returns a delta store (representing changes to the original store) instead of the updated store as part of the evaluation result. DSS not only preserves the advantages of pushdown control flow and abstract GC, but is able to exploit even greater precision benefits due to more aggressive GC. We provide both formal as well as empirical evidence for this claim. To prevent scalability issues, we propose the use of a form of store widening for DSS that still allows for flow sensitivity and remains compatible with abstract GC.

Finally, we show how an existing effect-driven worklist algorithm can be generalised to efficiently compute the fixpoint for DSS-based analyses (both with and without abstract GC). In order to further improve performance, we also design and implement a parallel version of this algorithm. Our empirical evaluation shows that this parallel worklist algorithm scales well on multi-core systems for both DSS-based analyses as well as other modular analyses.



```

to further
and imple-
nd. Our
parallel
-core
well

```

Abstract Garbage Collection for Efficient and Precise Abstract Interpretation

[illegible]

Noah Van Es

Promotor: Prof. Dr. Coen De Roover

Dissertation submitted in fulfilment of the requirement
for the degree of Doctor of Sciences

Abstract Garbage Collection for Efficient and Precise Abstract Interpretation

Noah Van Es

Dissertation submitted in fulfilment of the
requirement for the degree of Doctor of Sciences

23rd October 2026

Promotor:

Prof. Dr. Coen De Roover, Vrije Universiteit Brussel

Jury:

Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel (chair)

Prof. Dr. Jens Nicolay, Vrije Universiteit Brussel

Prof. Dr. Joost Vennekens, Vrije Universiteit Brussel

Dr. Raphaël Monat, Université de Lille

Dr. Caterina Urban, École Normale Supérieure

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering
Department of Computer Science
Software Languages Lab

Alle rechten voorbehouden. Niets van deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook, zonder voorafgaande schriftelijke toestemming van de auteur.

All rights reserved. No part of this publication may be reproduced in any form by print, photoprint, microfilm, electronic or any other means without permission from the author.

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel : +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

ISBN: 978-94-93461-88-8
NUR CODE: 993
THEMA: UMZT

Abstract

Abstract garbage collection is the application of garbage collection (GC) to an abstract interpreter. It is known that abstract GC significantly improves both the interpreter’s precision and performance. In this work, we focus on the efficient integration of abstract GC.

To maximise its precision and performance benefits, abstract GC needs to be applied frequently; at the same time, frequent applications bring about their own performance overhead. We propose a novel approach for abstract GC based on reference counting, which we call abstract reference counting. The approach is able to immediately eliminate all garbage, maximising abstract GC’s precision and performance benefits. Yet, it does not require frequent heap traversals, therefore adding no significant overhead. To ensure all garbage is eliminated, we augment abstract reference counting with cycle detection, counting references at the level of strongly connected components. The empirical evaluation shows that the resulting approach achieves both the optimal precision and significantly improved performance compared to existing approaches to abstract GC.

Second, we consider the integration of abstract GC for abstract interpreters that do not explicitly model continuations (such as abstract definitional interpreters). These offer extra precision and performance benefits by eliding an over-approximation of the continuation, and instead making use of recursive evaluation rules that inherit perfect stack precision from their defining language. Existing techniques either fail to exploit the full benefit of both abstract GC and such “big-step” abstract interpreters, and/or require complicated modifications to the interpreter. In addition, they all lack scalability, as abstract GC is not compatible with global store widening. We present delta store semantics (DSS), a novel yet simple approach to integrating abstract GC into abstract definitional interpreters. DSS returns a delta store (representing changes to the original store) instead of the updated store as part of the evaluation result. DSS not only preserves the advantages of pushdown control flow and abstract GC, but is able to exploit even greater precision benefits due to more aggressive GC. We provide both formal as well as empirical evidence for this claim. To prevent scalability issues, we propose the use of a form of store widening for DSS that still allows for flow sensitivity and remains compatible with abstract GC.

Finally, we show how an existing effect-driven worklist algorithm can be generalised to efficiently compute the fixpoint for DSS-based analyses (both with and without abstract GC). In order to further improve performance, we also design and implement a parallel version of this algorithm. Our empirical evaluation shows that this parallel worklist algorithm scales well on multi-core systems for both DSS-based analyses as well as other modular analyses.

Samenvatting

Abstracte garbage collection (GC) is de toepassing van garbage collection op abstracte interpreters. Het is geweten dat abstracte GC zowel de precisie als de performantie van een abstracte interpreter significant verbetert. In dit werk focussen we ons op de efficiënte integratie van abstracte GC.

Om de precisie- en performantiewinsten te maximaliseren moet abstracte GC regelmatig toegepast worden; tegelijkertijd brengen deze regelmatige toepassingen van GC hun eigen performantiekost met zich mee. Wij stellen een nieuwe aanpak voor abstracte GC voor die gebruik maakt van het tellen van referenties, en noemen deze aanpak abstract reference counting. Abstract reference counting kan meteen alle garbage elimineren, waardoor de precisie- en performantiewinsten gemaximaliseerd worden. Tegelijkertijd hoeft abstract reference counting het geheugen niet frequent te doorlopen, waardoor deze aanpak geen significante performantiekost toevoegt. Om ervoor te zorgen dat alle garbage verwijderd is, breiden we abstract reference counting uit met een algoritme om cycli te detecteren, en tellen we referenties op niveau van de sterk samenhangende componenten in het geheugen. De empirische evaluatie toont aan dat deze aanpak zowel de optimale precisie als een significant betere performantie behaalt in vergelijking met bestaande technieken voor abstracte GC.

Ten tweede bekijken we de integratie van abstracte GC voor abstracte interpreters die niet expliciet de huidige continuatie modelleren (zoals abstract definitional interpreters). Deze abstracte interpreters bieden extra voordelen op gebied van precisie en performantie door geen over-approximatie van de continuatie te modelleren, en in plaats daarvan gebruik te maken van recursieve evaluatieregels die perfecte stack precisie erven van de taal waarin ze zijn gedefinieerd. Bestaande technieken benutten niet de volledige voordelen van zowel abstracte GC als van deze “big-step” abstracte interpreters, en/of vereisen ingewikkelde aanpassingen aan de abstracte interpreter. Daarnaast ondervinden ze allemaal problemen met schaalbaarheid, aangezien abstracte GC niet compatibel is met global store widening. We presenteren delta store semantiek (DSS), een nieuwe maar eenvoudige aanpak om abstracte GC te integreren in abstract definitional interpreters. DSS geeft een delta store (die veranderingen aan de originele store voorstelt) terug in plaats van de gewijzigde store als deel van het evaluatieresultaat. DSS behoudt niet alleen de voordelen van pushdown control flow en abstracte GC, maar behaalt zelfs nog grotere precisiewinsten (vanwege de agressievere GC). We bevestigen dit zowel formeel als empirisch. Om schaalbaarheidsproblemen te voorkomen, stellen we het gebruik van een vorm van store widening voor DSS voor die zowel flow-sensitiviteit toelaat alsook compatibel is met abstracte GC.

Ten slotte tonen we aan hoe een bestaand effect-gedreven worklist algoritme gegeneraliseerd kan worden om efficiënt het fixpunt voor DSS-gebaseerde analyses (zowel met als zonder abstracte GC) te berekenen. Om de performantie nog meer te verbeteren, ontwerpen en implementeren we ook een parallelle versie van dit algoritme. Onze empirische evaluatie toont aan dat dit algoritme goed schaal op multi-core architecturen voor DSS-gebaseerde analyses en andere modulaire analyses.

Acknowledgements

First and foremost, I would like to thank my promotor, Coen De Roover, for having given me the opportunity to start a PhD, and for his guidance throughout the journey that followed. His feedback, insights and perspectives — along with the many discussions (and meetings) we had together — greatly helped to finish this work. Many times, Coen provided me with a map of the forest whenever I got stuck up high in some tree. I am also very grateful to have had a promotor who always made time for help when needed (even at less conventionally appropriate times). His continuous support, both in good times and in bad times, ultimately made it possible for me to complete this work.

I would also like to thank the members of my jury: Caterina Urban, Jens Nicolay, Joost Vennekens, Raphaël Monat and Wolfgang De Meuter. The constructive feedback and discussions they shared during the private defence greatly helped to uncover open issues and improve the quality of this work.¹

Next, I would like to thank my colleagues at the Software Languages Lab. I feel lucky to have ended up in the truly unique environment that is SOFT. Not just because of the interesting discussions that come up in our offices, meetings, or at the water dispenser, but also because of the moments where these discussions serve less academic value at Opinio, Pilar, and others.² A special thanks goes out to my *Monadic Analysis Friends*: Bram, Jens, Sarah and Quentin. Bram, for challenging every idea (and pulling me back from the clouds), and genuinely being a fun colleague with whom it has been very productive to explore ideas. Jens, for bringing colour to the group, taking care of our social needs, and helping crystallise the contours of our group efforts. Sarah, for bringing a lot into my office (and beyond); MAF will miss you. Quentin, for being the founding godfather of the group, and remaining there as an oracle we can turn to when things get really complicated. An additional round of thanks goes out to Bram, Sarah and Quentin for their feedback on certain chapters of this dissertation. I would also like to thank MAF-associate member Ruben, in particular for the enjoyable moments (and foot fetish stories) we shared on the rides back home, and SOFT-newbies Mats and Jolien for their frequent (but most welcome) visits to my office. Another big thank you goes to Giacomo, who during the last months of my writing took over the main responsibilities of the BUGATTI project (I know you always said it's "no big deal", but it surely was for me at the time).

¹Although I should still apologise for the excessive number of footnotes in this dissertation.

²Especially since both kinds of discussions are not confined to their respective locations.

Being at SOFT for a long time means that there are also a few ex-colleagues I want to thank. First of all, my former long-term office buddy Ward, with whom I shared the similarly long-term struggles in the PhD journey. While your daily afternoon runs and noise-cancelling headphones suggested otherwise, I enjoyed the company we shared at 10F728 — “*merci, Ward!*”. I also want to thank my former train buddies, Jonas and Kevin, for making the commutes more pleasant. Finally, I want to reserve a special thanks to Prof. em. Theo D’Hondt. His courses sparked my interest in language design and implementation, and he provided me with the first opportunities to explore these interests through a (bachelor) student job. Thank you for bringing me into SOFT!

Last but not least zijn er nog enkele belangrijke mensen in mijn omgeving die ik zou willen bedanken. Eerst en vooral, mijn mama, die mij altijd (al lang voordat ik aan dit doctoraat begon) doorheen *alles* heeft gesteund, mij de beste omgeving die ik kon wensen heeft gegeven om te groeien, en mij van jongs af aan heeft geleerd om ook tijdens de moeilijkere momenten door te zetten. Mijn papa, die de interesse in dit soort onderwerpen, alsook de drang om te ontdekken, op mij heeft overgebracht. Ons oma, voor alle steun en warmte die je mij doorheen het leven en tijdens mijn studies hebt gegeven, van de fruitsalades (met stevia!) die je me bracht op late studieavonden tot je wijsheden die me terug wisten te motiveren (“*Moed, want het moet!*”). Opa: ik wil je bedanken voor alles wat je mij hebt geleerd, voor je enthousiasme en steun die je mij hebt gegeven, en voor de ambitie die je mij hebt bijgebracht om aan dit werk te beginnen. Aan mijn schoonouders, Ingrid & Filip: bedankt om mij bij deze laatste loodjes extra te soigneren. Ook mijn vrienden, in het bijzonder Maarten & Mathias, en mijn kozijnen, Chris & Axel, wil ik bedanken om tijdens deze periode ook voldoende momenten van ontspanning in mijn leven te voorzien. Ten slotte wil ik mij richten tot mijn partner, Sarah: jouw steun was onmisbaar, zowel op de goede als op de minder mooie momenten van dit traject; ik blijf je dankbaar voor alles wat je de voorbije maanden voor mij hebt betekend — *forever*.

*“Je n’ai fait celle-ci plus longue que parce que je
n’ai pas eu le loisir de la faire plus courte.”*

— Blaise Pascal

Contents

1	Introduction	1
1.1	Research Context	1
1.1.1	The Importance of Abstract Garbage Collection	4
1.2	Problem Statement	7
1.3	Contributions	8
1.4	Supporting Publications	10
1.4.1	Other Publications	11
1.5	Outline of the Dissertation	13
2	Background	15
2.1	A Minimal Language λ_{ANF}	15
2.2	Concrete Semantics of λ_{ANF}	16
2.2.1	Concrete Garbage Collection	21
2.3	Abstract Semantics of λ_{ANF}	24
2.3.1	Abstract Domains	31
2.3.2	Abstract Garbage Collection	37
2.3.3	Abstract Counting	42
2.3.4	Abstract Store Widening	46
3	Abstract Reference Counting	49
3.1	Motivation	50
3.1.1	Problem: Application Policies for Abstract GC	50
3.1.2	Approach: Abstract Reference Counting	52
3.2	Reference Counting for Abstract GC	53
3.2.1	Abstract Interpretation with Reference Counting using $(\rightrightarrows_{\text{arc}})$	54
3.2.2	Properties of $(\rightrightarrows_{\text{arc}})$	59
3.3	Reclaiming Garbage Cycles	61
3.3.1	Artificial Cycles in $(\rightrightarrows_{\text{arc}})$	62
3.3.2	Abstract Reference Counting with Cycle Detection: $(\rightrightarrows_{\text{arc}++})$	63
3.3.3	Properties of $(\rightrightarrows_{\text{arc}++})$	69
3.4	Evaluation	72
3.4.1	Experimental Setup	72
3.4.2	RQ1: Impact of Cycle Detection	74

3.4.3	RQ2: Comparison to Existing Policies for Abstract GC . . .	76
3.5	Related Work	78
3.6	Conclusion	85
4	Delta Store Semantics	87
4.1	Motivation	88
4.1.1	Goal: The Best of Both Worlds	88
4.1.2	Approach: Delta Store Semantics	91
4.2	Abstract Definitional Semantics of λ_{ANF}	92
4.3	Delta Store Semantics	96
4.3.1	Delta Stores	97
4.3.2	Big-Step Abstract Semantics with Delta Stores for λ_{ANF} . .	98
4.4	Integrating Abstract GC	100
4.4.1	Replaying Delta Stores	100
4.4.2	Delta Store Semantics with Abstract GC	102
4.4.3	Comparison to Traditional Abstract GC	104
4.5	Integrating Flow-Sensitive Store Widening	106
4.5.1	The Need for Store Widening	106
4.5.2	Delta Store Semantics with Flow-Sensitive Store Widening	108
4.5.3	Impact on Abstract GC	111
4.6	Evaluation	112
4.6.1	Experimental Setup	112
4.6.2	RQ1: The Precision Benefit of Stackless Abstract GC . . .	114
4.6.3	RQ2: The Impact of Flow-Sensitive Store Widening	116
4.6.4	RQ3: The Impact of Abstract GC on Flow-Sensitive DSS .	117
4.7	Related Work	119
4.8	Conclusion	125
5	A Parallel Worklist Algorithm for DSS	127
5.1	Motivation	128
5.1.1	Problem: Parallel Analysis for Higher-Order Languages . .	129
5.1.2	Approach: A Parallel, Effect-Driven Worklist Algorithm . .	130
5.2	A Sequential Worklist Algorithm for DSS	131
5.2.1	Effect-Driven Worklist Algorithm	131
5.2.2	Instantiation for DSS	135
5.2.3	Instantiation for DSS-FS	138
5.3	A Parallel Worklist Algorithm for DSS	142
5.3.1	Parallel Effect-Driven Worklist Algorithm	142
5.3.2	Optimising for Parallel Efficiency	146

5.4	Evaluation	149
5.4.1	Experimental Setup	150
5.4.2	RQ1: Impact of the Effect-Driven Worklist Algorithm for DSS	152
5.4.3	RQ2: Parallel Efficiency for DSS	153
5.4.4	RQ3: Parallel Efficiency for MODF	156
5.5	Related Work	157
5.6	Conclusion	165
6	Conclusion and Future Work	167
6.1	Summary of the Dissertation	167
6.2	Contributions	172
6.3	Limitations & Future Work	174
6.4	Concluding Remarks	177
	Bibliography	179

List of Tables

3.1	Overview of the benchmark programs for Chapter 3	73
3.2	Comparison of performance and precision for abstract reference counting with and without cycle detection	75
3.3	Comparison of precision and performance of abstract reference counting with traditional approaches to abstract GC	76
4.1	Overview of the benchmark programs for Chapter 4	113
4.2	Comparison of precision and performance between small-step GCFA and big-step DSS with (stackless) abstract GC	115
4.3	Comparison of precision and performance between DSS with and without flow-sensitive store widening (DSS-FS and DSS, resp.) . .	116
4.4	Comparison of precision and performance between DSS-FS with and without abstract GC	118
5.1	Overview of the extra benchmarks used in Chapter 5	151
5.2	Comparison of time taken and total number of iterations between a naive and effect-driven fixpoint algorithm for DSS.	152
5.3	Comparison of time taken and total number of iterations between a naive and effect-driven fixpoint algorithm for DSS-FS.	153
6.1	Overview of the approaches discussed in this dissertation and their characteristics	171

List of Figures

1.1	Illustration of how abstract GC can prevent precision losses.	5
1.2	Illustration of how abstract GC can prevent redundant re-evaluations.	6
2.1	Grammar of the minimal higher-order language λ_{ANF}	15
2.2	State space of the concrete interpreter for λ_{ANF}	17
2.3	Transition rules of the concrete interpreter for λ_{ANF}	18
2.4	Program e_A (left) and the names of its subexpressions (right).	20
2.5	State space of the abstract interpreter for λ_{ANF}	25
2.6	Transition rules of the abstract interpreter for λ_{ANF}	30
2.7	Hasse diagram of the constant propagation lattice $\mathbf{CP}_{\text{Num}}$	35
2.8	Program e_E (left) and the names of its subexpressions (right).	45
3.1	States explored by the abstract interpreter for different GC policies.	52
3.2	The updated state space $\hat{\Sigma}_{\text{arc}}$ of the abstract interpreter with reference counting for λ_{ANF}	54
3.3	Transition relation $(\rightrightarrows_0)$ for abstract reference counting in λ_{ANF}	57
3.4	Example of artificial cycles arising due to abstraction.	63
3.5	Updated state space $\hat{\Sigma}_{\text{arc}}$ of the abstract interpreter using reference counting with cycle detection for λ_{ANF}	64
3.6	Comparison of overhead measured in time spent on GC per state for abstract reference counting with and without cycle detection	76
3.7	Comparison of overhead measured in time spent on GC per state for abstract reference counting and tracing abstract GC	77
5.1	Speedups for DSS and DSS-FS, each in a context-insensitive and a context-sensitive configuration for a varying number of workers.	154
5.2	Speedups for MODF, using both a context-insensitive and a context-sensitive configuration for a varying number of workers.	156

List of Acronyms

AAC	Abstracting Abstract Control
AAM	Abstracting Abstract Machines
ACC	ascending chain condition
ADI	Abstracting Definitional Interpreters
ANF	A-Normal Form
ARC	abstract reference counting
CFA	control flow analysis
CFG	control flow graph
CPS	continuation-passing style
DAG	directed acyclic graph
DSS	delta store semantics
DSS-FS	DSS with flow-sensitive store widening
GC	garbage collection
LLM	large language model
LOC	lines of code
SCC	strongly connected component
STM	Software Transactional Memory

1 Introduction

In this dissertation, we present several novel techniques to improve the precision and performance of static program analysis. Specifically, we focus on abstract interpretation, a powerful static analysis technique that can be used to reason about a program’s run-time behaviour by directly (over-)approximating its semantics. The main techniques we present in this dissertation involve improving the efficiency of existing approaches to abstract garbage collection, which can improve both the precision and performance of abstract interpreters by eliminating irrelevant information from their abstract states. In particular, we propose *abstract reference counting*, an alternative to the traditional tracing GC approach to abstract GC, and *delta store semantics*, a novel form of language semantics that allows an efficient integration of abstract GC for big-step abstract definitional interpreters. In addition, we also present a parallel worklist algorithm to efficiently compute the fixpoint of such analyses.

1.1 Research Context

Today’s society is increasingly reliant on software systems. As software development is inherently error-prone, errors in such software may have a major disruptive impact on society. As an example, in 2024, the CrowdStrike-related outage [6, 149] resulted in the crash of around 8.5 million Windows machines, which in turn caused major disruptions for airlines, hospitals, and other critical infrastructure. At the core of this outage was a software bug that caused an out-of-bounds access (specifically, reading the 21st element of an array of size 20), which in turn crashed the affected machines. Although the fix for this bug was relatively simple (i.e., adding a simple bounds check before accessing the array), the damage was already done by the time the fix was deployed, and the outage lasted for several days. This example stresses the importance of finding such errors early on in the development process (“shifting left”), a principle which is becoming increasingly recognised in modern software engineering.

One way in which such vulnerabilities can be detected early on is through static program analysis. Static analysis allows us to reason about the run-time behaviour of a program, without actually executing that program. As such, it

can for instance be used to detect potential occurrences of software bugs (such as the out-of-bounds access in the CrowdStrike-related outage) before the software is deployed. Unlike testing, static analysis reasons about all possible executions of a program, and can therefore be used to provide strong guarantees about the absence of certain classes of bugs (as it can conclude that the bug cannot occur in *any* possible execution path of the program). It is therefore frequently employed to provide such guarantees for safety-critical software in, for instance, aviation [19, 150, 82] (including space aviation [24, 22]), the automotive industry [80, 65] (as pushed by the ISO 26262 safety standard), and nuclear power plants [126].

The reasoning capabilities of static program analysis are not just limited to verifying the absence of bugs. In general, static analysis can be used to reason about any property of a program’s run-time behaviour. It therefore also has applications in other domains, such as program optimisation (e.g., to safely perform compiler optimisations that preserve program behaviour [176, 138]), program comprehension (e.g., to reduce a larger program to a *program slice* containing only expressions and statements relevant for a given program point of interest [178, 18]), or to automatically derive constraints for a program’s inputs [158]), and security (e.g., to verify the absence of exploitable security vulnerabilities [174, 20, 91] and information leaks [8, 9]).

The advent of large language models (LLMs) has further increased the need for static analysis. These LLMs are trained on large amounts of source code, and can generate code that is syntactically correct and compiles without errors; however, the generated code may still contain issues that are not immediately apparent and only manifest themselves at runtime. In fact, recent studies show that LLM-generated code often contains a significant number of bugs and vulnerabilities [75, 128, 127]. Static analysis can be used as a guardrail to verify the correctness of generated code [5, 117], providing guarantees about the absence of certain classes of bugs. Likewise, the growing popularity of LLM-generated code also increases the need for static analysis tools that aid developers with other issues such as program comprehension (as developers may struggle to understand the AI-generated code [159, 95, 3]) and program optimisation (as AI-generated code may suffer from performance issues [67, 130]).

However, the limits of static analysis are well known: Rice’s theorem (which can be seen as a generalisation of the halting problem) states that any non-trivial property of a program’s run-time behaviour is undecidable [134]. As a consequence, it is fundamentally impossible to design a static analysis for a non-trivial behavioural property that is completely accurate for any given program. In practical terms, this means analyses must trade accuracy for

computability by reasoning about an *approximation* of the program’s run-time behaviour. Over-approximating the program’s behaviour leads to *false positives*, as the analysis considers program behaviours that are not possible in any actual execution of the program. Under-approximating the program’s behaviour leads to *false negatives*, as the analysis fails to consider program behaviours that are possible in an actual execution of the program. An analysis is *sound* if it only over-approximates the program’s behaviour (and therefore only loses accuracy through false positives). An analysis is *complete* if it only under-approximates the program’s behaviour (and therefore only loses accuracy through false negatives). Due to Rice’s theorem, it is impossible to design a static analysis that is both sound and complete for any non-trivial property of a program’s run-time behaviour. In practice, most static analyses are designed to be sound, as this allows them to provide strong guarantees about the absence of certain classes of bugs (i.e., a bug is guaranteed to never occur in a program if it does not occur in an over-approximation of the program’s behaviour).

In order to be appreciated, a static analysis has to be sufficiently precise: studies show that a lack of precision (i.e., too many false alarms) is one of the main reasons why developers eschew the use of static analysis tools [76, 28, 140]. In general, a static analysis can be configured through several parameters to control its precision (i.e., the degree of (over-)approximation). Unfortunately, increasing the precision through these parameters often comes at a cost, as they generally also increase the analysis’ time and space complexity. As a result, an analysis that is sufficiently precise may fail to scale to larger programs (while studies show that a lack of performance also leads to developers not using static analysis [28, 43]), and an analysis that is sufficiently scalable may fail to be precise enough to be useful. A good balance between precision and scalability is often difficult to strike.

This Work: Static Analysis Through Abstract Interpretation. In this dissertation, we focus on the precision and performance of static analyses based on *abstract interpretation* [33, 105]. Abstract interpretation is a powerful approach to systematically construct (sound) static analyses for programs written in some programming language, starting from an interpreter for that language (following the language’s *concrete* operational semantics). The key idea is to systematically over-approximate (i.e., *abstract*) the behaviour of this “concrete” interpreter, resulting in an *abstract interpreter* (following the language’s *abstract semantics*). This approximation is done in such a way that the resulting abstract interpreter is always guaranteed to terminate (even when the concrete interpreter would enter an infinite loop); reasoning about the results of the execution of this abstract interpreter therefore yields a decidable (and sound) analysis. Specifically, the ap-

proximation usually entails finitising the heap, and replacing the concrete values used by the interpreter with abstract values coming from an *abstract domain* that is either finite, or for which it can be shown that the interpreter will only consider a finite number of values during its execution. To obtain a sound analysis, the abstract semantics is subsequently defined to consider all possible execution paths in case of non-determinism (which may arise from the use of abstract values).

A key advantage of abstract interpretation is that it supports a wide range of analyses (as the operational semantics can easily be extended to track the property of interest) and programming language features (as the operational semantics can easily handle advanced programming language concepts such as higher-order functions and first-class continuations). Additionally, an abstract interpreter offers various configuration parameters to control its precision and performance (such as the choice of a more or less precise abstract domain).

1.1.1 The Importance of Abstract Garbage Collection

One way to improve both the precision and performance of a static analysis is through *abstract garbage collection* [71, 106, 40] (abstract GC – also referred to as *memory localisation* [122] or *framing* [135, 97]). The idea of abstract GC is to eliminate irrelevant information from the memory representation of the abstract interpreter — specifically, information that is irrelevant to the current evaluation context. In particular, throughout its execution, an abstract interpreter typically maintains an approximation (i.e., an abstraction) of the heap, which maps (abstract) addresses to (abstract) values. Some bindings in the heap become unreachable during the execution of the abstract interpreter when their corresponding address is no longer (transitively) referenced from the current environment or stack of the interpreter, and can therefore be considered *garbage*. Similar to a concrete interpreter, an abstract interpreter can reclaim garbage by applying GC.

However, unlike garbage collection for a concrete interpreter, the primary purpose of abstract GC is not to free up memory (although this remains a secondary benefit). Instead, abstract GC is mainly used to improve the precision and scalability of the abstract interpreter. Specifically, we can identify two main advantages of garbage collection for an abstract interpreter.

Avoiding Precision Losses Due to Garbage Joins. In order to guarantee termination, the memory representation of an abstract interpreter is typically rendered finite by using an approximation of the heap with only a finite number of addresses. As the abstract interpreter needs to map a potentially infinite number of concrete addresses to a finite number of abstract addresses, the pigeonhole principle dictates that multiple concrete addresses inevitably end up being allocated

at the same abstract address. In order to soundly over-approximate the concrete heap, values allocated at the same address in the abstract heap are *joined* together. Such a join ensures that the resulting value over-approximates both of the joined values, but comes at the cost of precision. In general, such precision losses are unavoidable when using finite heap approximations.

Abstract GC can avoid some of these precision-detrimental joins. Without abstract GC, unreachable (i.e., reclaimable) addresses still hold on to (garbage) values. New allocations at these unreachable addresses need to join together the newly allocated values with these garbage values, leading to unnecessary precision losses. With abstract GC, the garbage values can be removed before new values are allocated at these addresses, thereby preventing these precision-detrimental joins. This situation is illustrated in Figure 1.1: without abstract GC (Figure 1.1a), a precision loss occurs at address $\hat{a}_3$, where the newly allocated value $\hat{v}_6$ is joined with the old garbage value $\hat{v}_5$; with abstract GC (Figure 1.1b), this address is reclaimed before the new allocation, and the join is avoided.

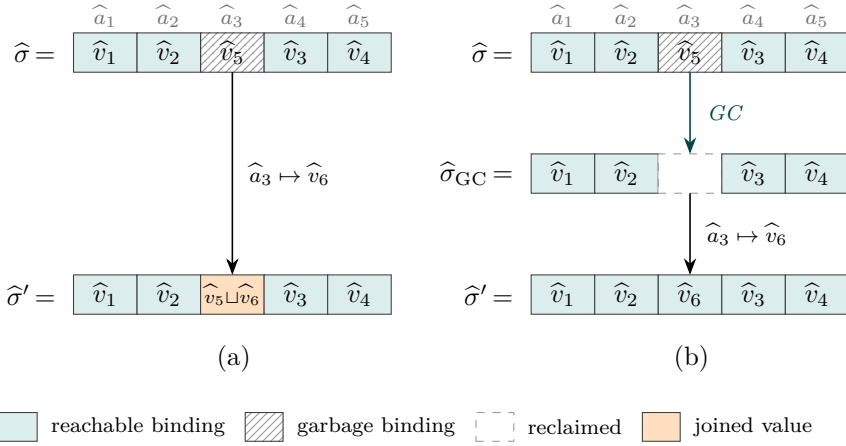


Figure 1.1: Illustration of how abstract GC can prevent precision losses.

In practice, these precision improvements brought by abstract GC also have a positive impact on the performance (and therefore scalability) of the abstract interpreter. The reason is that, without abstract GC, garbage is often easily propagated to other parts of the heap. At some point, the garbage may therefore influence the control-flow behaviour of the abstract interpreter, leading to the exploration of so-called *spurious paths* (i.e., program paths that never occur in a real execution of the program). Abstract GC prunes these garbage-induced spurious paths, therefore often significantly improving the interpreter's performance [106].

Avoiding the Reanalysis of Equivalent States. An abstract interpreter frequently needs to evaluate the same expression multiple times for different abstract heaps. In order to compute a fixpoint for evaluations where the same expression is recursively evaluated with the same heap, an abstract interpreter can employ some form of *cache* that stores the evaluation result of each expression for each of the different heaps it is encountered with. If the same expression is later evaluated with a heap that has already been encountered, it reuses the cached result. Otherwise, it re-evaluates the expression using the newly encountered heap.

Without abstract GC, some re-evaluations of the same expression are redundant, as two heaps may differ only in bindings that are irrelevant to the current evaluation context (i.e., in bindings that can be considered garbage). With abstract GC, such heaps would be considered equal, as removing all garbage keeps memory representations minimal. This increases the chances of evaluating an expression with the same heap, improving cache reuse (and therefore scalability). This situation is illustrated in Figure 1.2: without abstract GC (Figure 1.2a), the abstract interpreter needs to consider 5 different input heaps; when garbage bindings are removed from these heaps using abstract GC (Figure 1.2b), only two different heaps for which the expression needs to be evaluated remain.

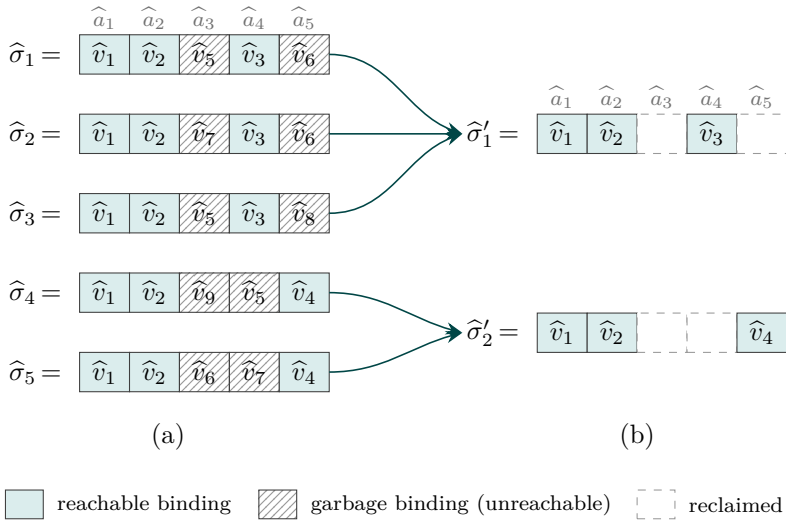


Figure 1.2: Illustration of how abstract GC can prevent redundant re-evaluations.

1.2 Problem Statement

The precision and performance benefits of abstract GC are widely recognised. However, the integration of abstract GC into an abstract interpreter is not always straightforward. We argue that most existing integrations (such as those presented in [105, 40]) fail to exploit the maximum precision and performance benefits of abstract GC. Specifically, in this dissertation, we focus on two key challenges.

Avoiding GC Overhead. As previously mentioned, abstract GC greatly improves the performance of the abstract interpreter, as it avoids the exploration of spurious paths and redundant re-evaluations. However, the application of abstract GC itself also brings about its own performance overhead, as each application of GC requires a full traversal of the heap to reclaim garbage. Crucially, the size of the heap is generally proportional to the size of the program: larger programs feature more allocation sites, which each use a different address in the heap. This makes abstract GC increasingly more expensive per application for larger programs.

In order to obtain its maximum benefit, abstract GC should also be applied *at every evaluation step* (as any surviving garbage may become reachable again after the address at which it resides is reused for a new allocation, resulting in an unrecoverable loss of precision). The combination of frequent applications and full heap traversals can easily become the performance bottleneck of the abstract interpreter. While the net effect of abstract GC on the interpreter’s performance is generally still positive, the benefit is significantly dragged down by this overhead. Practical implementations of abstract GC [106, 104, 73] therefore often choose to not apply abstract GC at every step, but only at certain points (e.g., at every function call). Such policies are however not ideal, as they come at the cost of precision, and only partially mitigate the negative impact of the GC overhead.

Integration With Big-Step Semantics. The application of concrete and abstract GC requires a *root set* of addresses, so that all addresses that can transitively be reached from this root set are not considered garbage (and thus survive GC). This root set typically comes from the evaluation environment and the current continuation (i.e., the “stack”) of the interpreter. When integrating abstract GC into a small-step abstract interpreter (e.g., as in [106, 105]), both the environment and continuation component are directly available.

Big-step abstract interpreters (e.g., as in [40]) offer several advantages over small-step abstract interpreters in terms of precision and performance. However, these interpreters do not explicitly model the continuation, and are instead built on recursive evaluation rules. The integration of abstract GC for such interpreters

hence becomes less trivial, as references from the continuation are required to correctly determine the GC root set (otherwise, GC may accidentally remove bindings that are required after the current evaluation).

Existing attempts to combine big-step abstract interpreters with abstract GC fail to do so effectively. For example, a straightforward approach [45, 40] is to modify the big-step abstract interpreter to explicitly pass along or recover the root set of addresses for the continuation. However, doing so is detrimental to both abstract GC (as these references are not needed for the current evaluation, and hence less garbage can be collected) and the interpreter’s scalability (as this may cause redundant re-evaluations of the same expression for different continuations). In summary, these attempts either fail to achieve the maximum precision and performance benefits when combining abstract GC with big-step semantics [45, 40, 53] and/or require complicated modifications to the abstract interpreter [45, 54].

1.3 Contributions

In this dissertation, we make the following contributions to overcome these challenges and obtain greater performance and precision benefits from abstract GC.

Abstract Reference Counting. We introduce a novel form of abstract GC based on reference counting instead of tracing GC, which we refer to as *abstract reference counting*. Using reference counting over tracing GC offers two important advantages. First, reference counting does not require full heap traversals; instead, it can directly reclaim unreachable addresses as soon as their reference count becomes zero. As a result, the performance of abstract reference counting is only sensitive to the amount of garbage that needs to be reclaimed, not to the size of the entire heap. Second, reference counting can reclaim all garbage immediately using simple bookkeeping, and hence does not require application heuristics that need to trade off precision (due to surviving garbage) for performance. These benefits eliminate the performance issues due to GC overhead.

We extend abstract reference counting with cycle detection, as naive reference counting cannot reclaim garbage cycles. Specifically, we integrate an incremental cycle detection algorithm into the abstract interpreter, and lift abstract reference counting to count references at the level of strongly connected components. We show that abstract reference counting with cycle detection results in a *garbage-free* abstract interpreter, as all garbage can be immediately reclaimed.

We provide an implementation of abstract reference counting as an extension of a small-step (AAM-style [105]) abstract interpreter in the MAF analysis frame-

work [165]. Our experiments confirm that abstract reference counting (with cycle detection) achieves the optimal precision for abstract GC, while significantly outperforming existing approaches to abstract (tracing) GC.

Delta Store Semantics (DSS). We introduce *delta store semantics* (DSS), a novel formulation of big-step abstract semantics where evaluation steps return minimal and composable delta stores (representing changes to a store) instead of complete updated stores. The main benefit of DSS is that it enables an efficient integration of abstract GC¹ (i.e., an integration that is able to preserve the maximum precision and performance benefits of both abstract GC and big-step abstract interpretation). For big-step abstract interpreters, we show that DSS with abstract GC achieves even better precision compared to an equivalent small-step interpreter with abstract GC (for programs without mutable variables). We formulate this claim as a theorem, and provide a mechanised proof in Rocq.

We also show how a new form of store widening can be integrated into DSS, further exploiting the minimal and compositional nature of delta stores. This form of store widening closely resembles the traditional notion of flow sensitivity, and still allows for strong updates and (a limited form of) abstract GC.

We formalise the base evaluation rules for DSS and the integration of abstract GC and flow-sensitive store widening. We provide an implementation of DSS (and these integrations) in MAF. Using this implementation, we conduct an empirical evaluation to measure the impact of abstract GC (compared to abstract GC for small-step interpreters) and flow-sensitive store widening for DSS.

A Parallel Worklist Algorithm for DSS. In order to compute a fixpoint for the big-step evaluation rules of DSS, we generalise an existing effect-driven worklist algorithm for a class of flow-insensitive, modular analyses. Specifically, we generalise this algorithm to support a larger class of analyses (including flow-sensitive ones), reformulate DSS as a modular analysis, and instantiate the generalised worklist algorithm to obtain an efficient fixpoint computation for DSS.

Subsequently, we show how this general worklist algorithm can be rendered parallel, further improving the scalability of the analysis. As the parallelisation is not specific to the instantiation of the worklist algorithm for DSS, this parallel worklist algorithm remains applicable to a wide class of modular analyses.

We provide an implementation of both a sequential and parallel version of this generalised worklist algorithm in MAF. Our experiments show that the effect-

¹Both abstract tracing GC and abstract reference counting are considered approaches to abstract GC. As our work on DSS and abstract reference counting form orthogonal contributions, we present each separately from the other, and therefore employ tracing GC for DSS.

driven worklist algorithm significantly outperforms a naive fixpoint algorithm, and that its parallel variant achieves significant speedups for both DSS and other existing modular analyses (such as ModF [119]).

1.4 Supporting Publications

The work presented in this dissertation builds upon several of our publications:

Van Es, N., Stiévenart, Q., and De Roover, C. (2019). Garbage-free abstract interpretation through abstract reference counting. In A. F. Donaldson, editor, *Proceedings of the 33rd European Conference on Object-Oriented Programming, LIPICs*, pages 10:1–10:33, London, United Kingdom. Schloss Dagstuhl - Leibniz-Zentrum für Informatik

This publication presents abstract reference counting as a novel, alternative approach to traditional abstract GC that does not require frequent heap traversals (and therefore achieves the optimal precision benefits of abstract GC with significantly lower overhead). Abstract reference counting is the topic of Chapter 3. Specifically, Chapter 3 extends the work of this publication by also taking into account strong updates, requiring modifications to the cycle detection algorithm.

Van Es, N., Van der Plas, J., Stiévenart, Q., and De Roover, C. (2020a). MAF: A framework for modular static analysis of higher-order languages. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 37–42, Adelaide, Australia. IEEE

This publication presents MAF (short for “Modular Analysis Framework”). MAF is a (Scala) framework that aids analysis developers in building modular analyses². Both the (small-step) AAM-style abstract interpreters of Chapter 3, as well as the (big-step) abstract definitional interpreters of Chapters 4 and 5 can be expressed in this modular analysis framework. Common to all these analyses is the effect-driven worklist algorithm that drives all modular analyses in MAF, which is generalised in this work and presented in Section 5.2. MAF itself is also a modular framework, in the sense that it allows analysis developers to easily build an analysis by composing different analysis components. All implementations of the work presented in this dissertation, as well as all experiments performed with those implementations for the evaluations presented in this dissertation, were conducted in MAF³.

²Following the notion of “modular” introduced by Cousot [35].

³Technically, with the exception of our work on abstract reference counting, which was built on top of Scala-AM [152], the predecessor of MAF.

Van Es, N., Stiévenart, Q., Van der Plas, J., and De Roover, C. (2020b). A parallel worklist algorithm for modular analyses. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 1–12, Adelaide, Australia. IEEE

This publication presents a parallel version of the effect-driven worklist algorithm used in MAF to further speed up the analysis. This parallelisation strategy is the central topic of Section 5.3 in this dissertation.

Van Es, N., Vandenbogaerde, B., and De Roover, C. (2025). Delta store semantics: Abstract garbage collection for abstract definitional interpreters. In H. Oh and Y. Sui, editors, *Proceedings of the 32nd International Static Analysis Symposium*, Lecture Notes in Computer Science, pages 386–421, Singapore. Springer

This publication presents delta store semantics (DSS), a novel approach to specifying (big-step) language semantics. The work shows how DSS makes it possible to integrate abstract GC into abstract interpreters that do not explicitly model the continuation component (i.e., “the stack”) while retaining the full benefits of both abstract GC and pushdown control flow. DSS is the topic of Chapter 4.

1.4.1 Other Publications

Several of our other publications do not directly support the work presented in this thesis. We list them below for the sake of completeness.

Van Es, N., Nicolay, J., Stiévenart, Q., D’Hondt, T., and De Roover, C. (2016). A performant Scheme interpreter in asm.js. In S. Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1944–1951, Pisa, Italy. ACM

Van Es, N., Stiévenart, Q., Nicolay, J., D’Hondt, T., and De Roover, C. (2017a). Implementing a performant Scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures*, **49**, 62–81

These articles were published before the start of the PhD research. They cover the work that was done as part of our Bachelor’s thesis, for which we investigated the integration of asm.js (the predecessor of today’s WebAssembly) to improve the performance of a (concrete) interpreter implemented in JavaScript.

Van Es, N., Vandercammen, M., and De Roover, C. (2017b). Incrementalizing abstract interpretation. In S. Demeyer, A. Parsai, G. Laghari, and B. van Bladel, editors, *Proceedings of the 16th edition of the BELgian-NEtherlands software eVOLution symposium*, CEUR Workshop Proceedings, pages 31–35, Antwerp, Belgium. CEUR-WS.org

This position paper summarises the main findings of the work that was done as part of our Master’s thesis. This work shares the same broader research field (abstract interpretation, with a focus on efficiency) as the work presented in this dissertation. Specifically, it proposes a technique to render the computation performed by a small-step (AAM-style) abstract interpreter incremental, in the sense that a full re-analysis of the whole program can be avoided upon incremental changes in the program (i.e., such an incremental analysis attempts to reuse previous analysis results and to recompute only what is affected by the change).

Van der Plas, J., Stiévenart, Q., Van Es, N., and De Roover, C. (2020). Incremental flow analysis through computational dependency reification. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–36, Adelaide, Australia. IEEE

This paper presents another strategy to render abstract interpretation incremental. It makes use of the same effect-driven worklist algorithm as the one that is at the centre of Chapter 5, and exploits its reified dependencies to determine which parts of the analysis need to be invalidated and recomputed.

Stiévenart, Q., Van Es, N., Van der Plas, J., and De Roover, C. (2021). A parallel worklist algorithm and its exploration heuristics for static modular analyses. *Journal of Systems and Software*, **181**, 111042

This journal article is an extension of the work on our parallel worklist algorithm (as presented in Section 5.1.2). Specifically, it extends our work by investigating the impact of several exploration heuristics (which determine the order in which components are scheduled for analysis) on the parallel efficiency of the algorithm.

Verbelen, S., Vandenbogaerde, B., Van der Plas, J., Van Es, N., and De Roover, C. (2024). Abstract slicing to improve the speed of static program analysis. In G. Perrouin, B. Vanderose, and X. Devroey, editors, *Proceedings of the 23rd Belgium-Netherlands Software Evolution Workshop*, CEUR Workshop Proceedings, pages 134–145, Namur, Belgium. CEUR-WS.org

This paper explores the use of abstract slicing (an advanced form of program slicing where only dependencies that actually impact the property of interest are

taken into account) as a way to speed up static analysis. Specifically, the idea is to reduce the size of the program under analysis using abstract slicing in order to improve the performance of a subsequent abstract interpretation of that program.

Vandenbogaerde, B., Verbelen, S., Van Es, N., and De Roover, C. (2025). Monarch: A modular framework for abstract definitional interpreters in Haskell. In H. Oh and Y. Sui, editors, *Proceedings of the 32nd International Static Analysis Symposium*, Lecture Notes in Computer Science, pages 357–385, Singapore. Springer

This paper presents the MONARCH analysis framework for abstract definitional interpreters (i.e., like those used in Chapters 4 and 5). MONARCH is the successor of MAF. The key novelty is that MONARCH – being implemented in Haskell (instead of Scala in the case of MAF) – uses a MONadic ARCHitecture to compose analysis components (using monad transformer stacks), whereas MAF uses a mix of monadic effect polymorphism and trait composition for this purpose.

1.5 Outline of the Dissertation

The remainder of this dissertation is structured as follows:

Chapter 2: Background. This chapter introduces all the supporting material. Specifically, it introduces a minimal language λ_{ANF} , for which all formalisations in this dissertation are formulated. It first introduces the concrete (small-step) operational semantics for λ_{ANF} as an abstract machine. Next, it systematically abstracts these concrete semantics (following the AAM approach to abstract interpretation [105]) to obtain the abstract semantics of λ_{ANF} . In doing so, it also explains the concepts of abstract domains, abstract garbage collection, abstract counting (for strong updates) and (global) store widening, and shows how these can be integrated into the abstract semantics for λ_{ANF} .

Chapter 3: Abstract Reference Counting. This chapter presents a novel, alternative approach to abstract GC, which we refer to as *abstract reference counting*. It shows how traditional abstract GC, based on tracing GC (as presented in Chapter 2), faces a tradeoff in its application policy: frequent applications are necessary to eliminate all garbage (and achieve the optimal precision for abstract GC), yet these same frequent applications bring about a significant performance overhead (due to frequent full heap traversals). Abstract reference counting, on the other hand, can collect all garbage continuously with only constant-time overhead. This chapter shows how abstract reference counting is integrated into the

(small-step) abstract semantics presented in Chapter 2. In this chapter, we also focus on cyclic garbage, which naive reference counting cannot reclaim, and which we show occurs more frequently in an abstract interpreter. Therefore, this chapter also presents a form of abstract reference counting augmented with cycle detection (so that references can be counted at the level of strongly connected components), which is able to collect all garbage (resulting in a *garbage-free* abstract interpreter).

Chapter 4: Delta Store Semantics. In this chapter, we focus on the integration of abstract GC into big-step (definitional) abstract interpreters (which do not model the stack). We translate the small-step semantics for λ_{ANF} into equivalent big-step abstract semantics, explain how existing approaches integrate abstract GC in such semantics, and highlight why they fall short of the full benefits of abstract GC and pushdown control flow. Next, we present a novel approach to express the (abstract) big-step semantics of a language using delta stores, and formulate such *delta store semantics* (DSS) for λ_{ANF} . The key point is that DSS allows for a more efficient integration of abstract GC, in that it can preserve all benefits of pushdown control flow and exploit even greater benefits from abstract GC. In this chapter, we also focus on improving scalability through store widening, which is generally considered incompatible with abstract GC. Therefore, this chapter also presents a novel form of store widening which preserves flow sensitivity and still allows for abstract GC, and integrates this into DSS for λ_{ANF} .

Chapter 5: A Parallel Worklist Algorithm for DSS. In this chapter, we focus on how an analysis using delta store semantics can efficiently be computed (i.e., how a fixpoint over the DSS evaluation rules can efficiently be computed). We first generalise an existing effect-driven worklist algorithm to compute such a fixpoint, and show how it can effectively be applied to DSS for λ_{ANF} . Next, we render the computation more efficient by designing a parallel version of this worklist algorithm, which is then again applied to DSS and other modular analyses.

Chapter 6: Conclusion and Future Work. We conclude with a chapter where we provide a summary of the work presented in this dissertation, recapitulating our main contributions. We also provide some potential directions for future work, such as the integration of abstract reference counting into DSS, an extension of our work to relational domains, and the adoption of adaptive analysis techniques.

2 Background

In this chapter, we introduce a minimal language λ_{ANF} . Although we implement and evaluate all techniques for a subset of R5RS Scheme, λ_{ANF} is used throughout this dissertation to formalise these techniques more succinctly. We first define its concrete small-step semantics, and extend them with concrete (tracing) garbage collection. Similarly, we define the corresponding abstract semantics, and extend them with abstract (tracing) garbage collection and abstract counting. Finally, we discuss abstract store widening, a common optimisation to aid termination and improve the overall performance of the abstract interpreter.

2.1 A Minimal Language λ_{ANF}

Figure 2.1 defines the syntax of a higher-order language λ_{ANF} , based on the λ -calculus in A-Normal Form (ANF)¹ [48]. ANF is a syntactic form that restricts operators and operands to *atomic expressions* ae which can be evaluated “atomically”, i.e., in a single step without impacting program state. This restriction allows for cleaner formal semantics: only **let**-expressions need to account for sequenced evaluations of other non-atomic expressions; all other expressions only contain atomic expressions, greatly simplifying their evaluation rules.

$e \in \text{Exp} ::= ae$	[ATOM]	$f, ae \in \text{AExp} ::= x \mid n \mid lam$
$f(ae)$	[CALL]	$lam \in \text{Lam} ::= \lambda x.e$
$op(ae_1, ae_2)$	[BINOP]	$x \in \text{Var}$ (a set of identifiers)
$\text{let } x = e_1 \text{ in } e_2$	[LET]	$op \in \text{Op}$ (a set of operators)
$\text{set } x := ae$	[SET]	$n \in \mathbb{N}$

Figure 2.1: Grammar of the minimal higher-order language λ_{ANF} .

¹Note that the syntactical specification for λ_{ANF} in Figure 2.1 is slightly less restrictive compared to the conventional formulation of ANF, as we allow any other expression e (i.e., including other **let**-expressions) on the right-hand side of **let**-expressions.

Additionally, we extend the language with variable assignment (SET) to model mutable references in the language. Unlike most minimal λ -calculus formulations, we also explicitly include numerical literals in our language, along with some built-in binary operators to work with these numbers (BINOP).

We assume that every expression e (and atomic expression ae) carries a unique label ℓ . We omit these labels for brevity, and only assume that they exist to preserve the ability to differentiate multiple occurrences of the same expression.

In this work, we target (a subset of) R5RS Scheme [1], a dynamically typed, higher-order language. Since any given program for this target language can automatically be converted into an equivalent A-Normal Form (ANF) program with mutable references, this simplification is purely cosmetic and without loss of generality. The main R5RS feature excluded by our subset is support for first-class continuations (through the `call/cc` primitive). We discuss this limitation further in Section 6.3, where we highlight the implications of adding support for `call/cc` for the abstract interpreters presented in this dissertation.

2.2 Concrete Semantics of λ_{ANF}

We formally define the concrete semantics of λ_{ANF} by presenting its concrete *operational* semantics. The operational semantics is defined using a small-step interpreter, formulated as an abstract machine. In what follows, we systematically describe the design of this abstract machine.

State Space. Figure 2.2 describes the state space Σ for the concrete interpreter. A state ς consists of a control component (either an evaluation of expression e under environment ρ , a call to a closure (lam, ρ) on an argument value v , or a resulting value v), a store σ (mapping addresses to values, to model the “heap”), a continuation store σ_k (mapping continuation addresses to continuations, to model the “stack”), the current continuation address a_k (pointing to “the top of the stack”) and a natural number n (representing the next fresh address to be allocated). In λ_{ANF} , the only values are closures (which pair a lambda expression with its lexical environment) and numbers; the only continuations in λ_{ANF} are those arising from the evaluation of a `let`-expression (which consists of the variable to which the resulting value should be bound and the next expression, environment, and continuation address to continue the evaluation). For a concrete interpreter, addresses (and continuation addresses) come from the infinite set $\mathbb{N}$ to ensure that each allocation can be given a fresh address.

$$\begin{array}{ll}
 \varsigma \in \Sigma ::= \langle c, \sigma, \sigma_k, a_k, n \rangle & \rho \in \text{Env} = \text{Var} \rightarrow \text{Addr} \\
 c \in \text{Control} ::= \mathbf{ev}(e, \rho) & \sigma \in \text{Store} = \text{Addr} \rightarrow \text{Val} \\
 \quad | \mathbf{call}(lam, \rho, v) & \sigma_k \in \text{KStore} = \text{KAddr} \rightarrow \text{Kont} \\
 \quad | \mathbf{val}(v) & v \in \text{Val} ::= \mathbf{clo}(lam, \rho) \mid \mathbf{num}(n) \\
 \kappa \in \text{Kont} ::= \mathbf{letk}(x, e, \rho, a_k) & a \in \text{Addr} = \mathbb{N} \\
 l \in \text{Loc} = \text{Addr} \cup \text{KAddr} & a_k \in \text{KAddr} = \mathbb{N} \cup \{\mathbf{a}_{\text{halt}}\}
 \end{array}$$

 Figure 2.2: State space of the concrete interpreter for λ_{ANF} .

For a state ς , we implicitly assume subscripted notations for its components so that $\varsigma = \langle c_\varsigma, \sigma_\varsigma, \sigma_{k\varsigma}, a_{k\varsigma}, n_\varsigma \rangle$. For (partial) maps, $[]$ denotes the empty map, while $m[a \mapsto b]$ extends m so that $m[a \mapsto b](a) = b$ and $m[a \mapsto b](x) = m(x)$ for $x \neq a$.

A store σ supports three main operations: *lookup*, *extend* (adding a new binding to σ) and *update* (modifying an existing binding in σ). We introduce helper functions to handle these operations, currently defined as follows:

$$\text{extend}(\sigma, a, v) = \text{update}(\sigma, a, v) = \sigma[a \mapsto v] \quad \text{lookup}(\sigma, a) = \sigma(a)$$

Transition Relation. Atomic expressions can be evaluated in a single step, without making any modifications to the store. We first introduce an auxiliary function $\mathcal{A} : AExp \times \text{Env} \times \text{Store} \rightarrow \text{Val}$ to evaluate atomic expressions:

$$\begin{array}{ll}
 \mathcal{A}(x, \rho, \sigma) = \text{lookup}(\sigma, \rho(x)) & [\text{A-VAR}] \\
 \mathcal{A}(lam, \rho, \sigma) = \mathbf{clo}(lam, \rho) & [\text{A-LAM}] \\
 \mathcal{A}(n, \rho, \sigma) = \mathbf{num}(n) & [\text{A-NUM}]
 \end{array}$$

A variable is atomically evaluated by looking for its address in the environment, and then looking up that address in the store (A-VAR). A lambda is atomically evaluated by pairing it with the current environment into a closure value (A-LAM). A numerical literal is atomically evaluated to a numerical value (A-NUM).

The evaluation rules of the concrete interpreter for λ_{ANF} are given using a small-step transition relation $(\rightarrow) \subseteq \Sigma \times \Sigma$, describing how the interpreter steps from one state to the next one. Figure 2.3 defines the transition rules for $(\rightarrow)$.

$$\begin{array}{c}
 \frac{v = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(ae, \rho), \sigma, \sigma_k, a_k, n \rangle \rightarrow \langle \mathbf{val}(v), \sigma, \sigma_k, a_k, n \rangle} \quad [\text{E-ATOM}] \\
 \\
 \frac{a = \rho(x) \quad v = \mathcal{A}(ae, \rho, \sigma) \quad \sigma' = \text{update}(\sigma, a, v)}{\langle \mathbf{ev}(\text{set } x := ae, \rho), \sigma, \sigma_k, a_k, n \rangle \rightarrow \langle \mathbf{val}(v), \sigma', \sigma_k, a_k, n \rangle} \quad [\text{E-SET}] \\
 \\
 \frac{\text{clo}(\text{lam}, \rho') = \mathcal{A}(f, \rho, \sigma) \quad v = \mathcal{A}(ae, \rho, \sigma)}{\langle \mathbf{ev}(f(ae), \rho), \sigma, \sigma_k, a_k, n \rangle \rightarrow \langle \mathbf{call}(\text{lam}, \rho', v), \sigma, \sigma_k, a_k, n \rangle} \quad [\text{E-CALL}] \\
 \\
 \frac{a = \text{alloc}(x, \varsigma) \quad \rho' = \rho[x \mapsto a] \quad \sigma' = \text{extend}(\sigma, a, v)}{\underbrace{\langle \mathbf{call}(\lambda x. e, \rho, v), \sigma, \sigma_k, a_k, n \rangle}_{\varsigma} \rightarrow \langle \mathbf{ev}(e, \rho'), \sigma', \sigma_k, a_k, n + 1 \rangle} \quad [\text{E-APP}] \\
 \\
 \frac{\text{num}(n_1) = \mathcal{A}(ae_1, \rho, \sigma) \quad \text{num}(n_2) = \mathcal{A}(ae_2, \rho, \sigma) \quad v = \text{num}(\mathcal{I}(op, n_1, n_2))}{\langle \mathbf{ev}(op(ae_1, ae_2), \rho), \sigma, \sigma_k, a_k, n \rangle \rightarrow \langle \mathbf{val}(v), \sigma, \sigma_k, a_k, n \rangle} \quad [\text{E-BINOP}] \\
 \\
 \frac{a'_k = \text{alloc}_k(e_1, \varsigma) \quad \kappa = \text{letk}(x, e_2, \rho, a_k) \quad \sigma'_k = \sigma_k[a'_k \mapsto \kappa]}{\underbrace{\langle \mathbf{ev}(\text{let } x = e_1 \text{ in } e_2, \rho), \sigma, \sigma_k, a_k, n \rangle}_{\varsigma} \rightarrow \langle \mathbf{ev}(e_2, \rho), \sigma, \sigma'_k, a'_k, n + 1 \rangle} \quad [\text{E-LET}] \\
 \\
 \frac{\text{letk}(x, e, \rho, a'_k) = \sigma_k(a_k) \quad a = \text{alloc}(x, \varsigma) \quad \rho' = \rho[x \mapsto a] \quad \sigma' = \text{extend}(\sigma, a, v)}{\underbrace{\langle \mathbf{val}(v), \sigma, \sigma_k, a_k, n \rangle}_{\varsigma} \rightarrow \langle \mathbf{ev}(e, \rho'), \sigma', \sigma_k, a'_k, n + 1 \rangle} \quad [\text{E-CONTINUE}]
 \end{array}$$

 Figure 2.3: Transition rules of the concrete interpreter for λ_{ANF} .

Rule E-ATOM immediately evaluates an atomic expression to a resulting value using the auxiliary function $\mathcal{A}$. Rule E-SET mutates a variable binding by updating the address of that variable in the store. Rule E-CALL evaluates a function call by atomically evaluating the operator and operand, resulting in a **call** state. Consequently, rule E-APP applies the closure to the argument of the **call** state by binding the function parameter to the argument value and then stepping into the evaluation of the function body. Rule E-BINOP evaluates a binary operator application by atomically evaluating both operands and subsequently applying the oper-

ator to those operand values. We assume a function $\mathcal{I} : \text{Op} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ exists to correctly interpret such built-in operators (where e.g., $\mathcal{I}(\text{plus}, n_1, n_2) = n_1 + n_2$). Rule E-LET evaluates a **let**-expression by first stepping into the evaluation of its right-hand side expression and installing a continuation so that the result of this evaluation is later bound to the variable introduced by the **let**-expression before continuing with the evaluation of its body; the latter happens in the E-CONTINUE rule, as there is only one kind of continuation in λ_{ANF} .

In a concrete interpreter, we need a fresh (i.e., unused) address for each new variable (or continuation) binding that is allocated in the store. Therefore, we define the corresponding allocation functions **alloc** and **alloc_k** as follows:

$$\text{alloc}(x, \varsigma) = n_\varsigma \qquad \text{alloc}_k(e, \varsigma) = n_\varsigma$$

We write $\varsigma_a \xrightarrow{n} \varsigma_b$ for a sequence of n steps using $(\rightarrow)$ (i.e., when $\varsigma_a = \varsigma_0 \rightarrow \varsigma_1 \rightarrow \dots \rightarrow \varsigma_n = \varsigma_b$). Similarly, we define $(\xrightarrow{*})$ as the reflexive, transitive closure of $(\rightarrow)$. Note that for a concrete interpreter, this transition relation is deterministic (i.e., if $\varsigma_0 \xrightarrow{n} \varsigma_1$ and $\varsigma_0 \xrightarrow{n} \varsigma_2$ then $\varsigma_1 = \varsigma_2$), and hence $(\rightarrow)$ is a (partial) function.

Program Semantics. The concrete interpreter for λ_{ANF} evaluates a program by first *injecting* it into an initial state, and then repeatedly applying the transition relation $(\rightarrow)$ until the final program state is reached (meaning no further applications of $(\rightarrow)$ to that state are possible). The initial state for a given program e is constructed using a function **inject**, defined as:

$$\text{inject}(e) = \langle \mathbf{ev}(e, []), [], [], a_{\text{halt}}, 0 \rangle$$

where a_{halt} is a special address which is never bound in the continuation store. When program evaluation reaches its final value, there are no further continuations at a_{halt} , and hence the E-CONTINUE rule cannot be applied any further. Formally, we can define an evaluation function $\text{eval} : \text{Exp} \rightarrow \text{Val}$:

$$\text{eval}(e) = v \iff \text{inject}(e) \xrightarrow{*} \langle \mathbf{val}(v), -, -, a_{\text{halt}}, - \rangle$$

We define the *collecting semantics* of a program as the set of all program states that are reached during the evaluation of that program. That is, we can define a function $S : \text{Exp} \rightarrow \mathcal{P}(\Sigma)$ as follows:

$$S(e) = \{\varsigma \in \Sigma \mid \text{inject}(e) \xrightarrow{*} \varsigma\}$$

Note that computing $S(e)$ is more general than computing $\text{eval}(e)$, as the latter (if it exists) is contained within some state in $S(e)$. By computing $S(e)$, we can also inspect all the intermediate states that are reached during the evaluation of e , and as such $S(e)$ can be used to reason about the program's behaviour during evaluation. Furthermore, by computing a set of states, it also allows for the modelling of non-determinism. While the transition relation ($\rightarrow$) for our simple λ_{ANF} language is deterministic, it could easily be extended to also include nondeterministic language features (e.g., by including a **random** function or by allowing for user input). In those cases, $S(e)$ would include all possible executions of the program e . Our transition relation will also be rendered nondeterministic when we introduce the abstract semantics later on (cf. Section 2.3).

Unfortunately, when e does not terminate, or when non-determinism is considered (e.g., due to unknown user input), the set $S(e)$ may not be computable for a given program e , as it can contain an infinite number of states in Σ .

Example. We illustrate how an expression in λ_{ANF} is evaluated using the concrete semantics for the example program shown in Figure 2.4 (which serves as a running example in this chapter). This program defines an identity function id and calls this function twice, defining a function f that increments its argument by 1, a function g that doubles its argument, and finally calls g with the argument 10.

$let\ id = \lambda x.x\ in$	$e_A = let\ id = \lambda x.x\ in\ e_B$
$let\ f = id(\lambda a.plus(a, 1))\ in$	$e_B = let\ f = id(\lambda a.plus(a, 1))\ in\ e_C$
$let\ g = id(\lambda b.plus(b, b))\ in$	$e_C = let\ g = id(\lambda b.plus(b, b))\ in\ e_D$
$g(10)$	$e_D = g(10)$

Figure 2.4: Program e_A (left) and the names of its subexpressions (right).

Evaluating this program using the concrete semantics (i.e., using the concrete transition relation ($\rightarrow$)) results in the following sequence of states:

$$\begin{aligned}
 \text{inject}(e_A) &= \varsigma_0 = \langle \mathbf{ev}(e_A, []), [], a_{\text{halt}}, 0 \rangle \\
 &\rightarrow \varsigma_1 = \langle \mathbf{ev}(\lambda x.x, []), [], \sigma_k^1, 0, 1 \rangle && [\text{E-LET}] \\
 &\rightarrow \varsigma_2 = \langle \mathbf{val}(v_{id}), [], \sigma_k^1, 0, 1 \rangle && [\text{E-ATOM}] \\
 &\rightarrow \varsigma_3 = \langle \mathbf{ev}(e_B, \rho_1), \sigma^1, \sigma_k^1, a_{\text{halt}}, 2 \rangle && [\text{E-CONTINUE}] \\
 &\rightarrow \varsigma_4 = \langle \mathbf{ev}(id(\lambda a.plus(a, 1)), \rho_1), \sigma^1, \sigma_k^2, 2, 3 \rangle && [\text{E-LET}]
 \end{aligned}$$

$$\begin{aligned}
 \rightarrow \varsigma_5 &= \langle \mathbf{call}(\lambda x.x, [], v_f), \sigma^1, \sigma_k^2, 2, 3 \rangle & [\text{E-CALL}] \\
 \rightarrow \varsigma_6 &= \langle \mathbf{ev}(x, [x \mapsto 3]), \sigma^2, \sigma_k^2, 2, 4 \rangle & [\text{E-APP}] \\
 \rightarrow \varsigma_7 &= \langle \mathbf{val}(v_f), \sigma^2, \sigma_k^2, 2, 4 \rangle & [\text{E-ATOM}] \\
 \rightarrow \varsigma_8 &= \langle \mathbf{ev}(e_C, \rho_2), \sigma^3, \sigma_k^2, a_{\text{halt}}, 5 \rangle & [\text{E-CONTINUE}] \\
 \rightarrow \varsigma_9 &= \langle \mathbf{ev}(id(\lambda b.\text{plus}(b, b)), \rho_2), \sigma^3, \sigma_k^3, 5, 6 \rangle & [\text{E-LET}] \\
 \rightarrow \varsigma_{10} &= \langle \mathbf{call}(\lambda x.x, [], v_g), \sigma^3, \sigma_k^3, 5, 6 \rangle & [\text{E-CALL}] \\
 \rightarrow \varsigma_{11} &= \langle \mathbf{ev}(x, [x \mapsto 6]), \sigma^4, \sigma_k^3, 5, 7 \rangle & [\text{E-APP}] \\
 \rightarrow \varsigma_{12} &= \langle \mathbf{val}(v_g), \sigma^4, \sigma_k^3, 5, 7 \rangle & [\text{E-ATOM}] \\
 \rightarrow \varsigma_{13} &= \langle \mathbf{ev}(e_D, \rho_3), \sigma^5, \sigma_k^3, a_{\text{halt}}, 8 \rangle & [\text{E-CONTINUE}] \\
 \rightarrow \varsigma_{14} &= \langle \mathbf{call}(\lambda b.\text{plus}(b, b), \rho_2, \mathbf{num}(10)), \sigma^5, \sigma_k^3, a_{\text{halt}}, 8 \rangle & [\text{E-CALL}] \\
 \rightarrow \varsigma_{15} &= \langle \mathbf{ev}(\text{plus}(b, b), \rho_4), \sigma^6, \sigma_k^3, a_{\text{halt}}, 9 \rangle & [\text{E-APP}] \\
 \rightarrow \varsigma_{16} &= \langle \mathbf{val}(\mathbf{num}(20)), \sigma^6, \sigma_k^3, a_{\text{halt}}, 9 \rangle & [\text{E-BINOP}]
 \end{aligned}$$

$$\begin{aligned}
 \text{where } \rho_1 &= [id \mapsto 1] & v_{id} &= \mathbf{clo}(\lambda x.x, []) \\
 \rho_2 &= [id \mapsto 1, f \mapsto 4] & v_f &= \mathbf{clo}(\lambda a.\text{plus}(a, 1), \rho_1) \\
 \rho_3 &= [id \mapsto 1, f \mapsto 4, g \mapsto 7] & v_g &= \mathbf{clo}(\lambda b.\text{plus}(b, b), \rho_2) \\
 \rho_4 &= [id \mapsto 1, f \mapsto 4, b \mapsto 8] \\
 \kappa_0 &= \mathbf{letk}(id, e_B, [], a_{\text{halt}}) & \sigma_k^1 &= [0 \mapsto \kappa_0] \\
 \kappa_1 &= \mathbf{letk}(f, e_C, \rho_1, a_{\text{halt}}) & \sigma_k^2 &= \sigma_k^1[2 \mapsto \kappa_1] \\
 \kappa_2 &= \mathbf{letk}(g, e_D, \rho_2, a_{\text{halt}}) & \sigma_k^3 &= \sigma_k^2[5 \mapsto \kappa_2] \\
 \sigma^1 &= [1 \mapsto v_{id}] & \sigma^4 &= \sigma^3[6 \mapsto v_g] \\
 \sigma^2 &= \sigma^1[3 \mapsto v_f] & \sigma^5 &= \sigma^4[7 \mapsto v_g] \\
 \sigma^3 &= \sigma^2[4 \mapsto v_f] & \sigma^6 &= \sigma^5[8 \mapsto \mathbf{num}(10)]
 \end{aligned}$$

The final state ς_{16} holds the resulting value of the program, i.e., $\mathbf{eval}(e_A) = \mathbf{num}(20)$. The collecting semantics for this program, which is equal to the set of all states reached during its evaluation, is $S(e_A) = \{\varsigma_i\}_{i=0}^{16}$.

2.2.1 Concrete Garbage Collection

We now add a garbage collector to this concrete interpreter. While garbage collection does not impact the results of a concrete interpreter, it is useful to reduce the sizes of the stores σ_ς and σ_{k_ς} that need to be stored in every state ς (therefore reducing the memory required to compute $S(e)$). The garbage collector considered here is called a *tracing* garbage collector, in the sense that it traverses

the entire store to keep all bindings that are still transitively reachable from some given root set of addresses [11, 78]. This root set, for a given state ς , consists of all addresses that are directly reachable and necessary to continue evaluation for ς .

We employ notation and definitions similar to those used by Might and Shivers [106]. First, we define a family of auxiliary functions $\mathcal{T}_X : X \rightarrow \mathcal{P}(\text{Loc})$ that return all addresses that are referenced directly from a given X .

$$\begin{array}{ll}
 \mathcal{T}_\Sigma(\langle c, \sigma, \sigma_k, a_k, n \rangle) = \mathcal{T}_{\text{Control}}(c) \cup \{a_k\} & \mathcal{T}_{\text{Control}}(\mathbf{ev}(e, \rho)) = \mathcal{T}_{\text{Env}}(\rho) \\
 \mathcal{T}_{\text{Control}}(\mathbf{call}(lam, \rho, v)) = \mathcal{T}_{\text{Env}}(\rho) \cup \mathcal{T}_{\text{Val}}(v) & \mathcal{T}_{\text{Control}}(\mathbf{val}(v)) = \mathcal{T}_{\text{Val}}(v) \\
 \mathcal{T}_{\text{Val}}(\mathbf{clo}(lam, \rho)) = \mathcal{T}_{\text{Env}}(\rho) & \mathcal{T}_{\text{Val}}(\mathbf{num}(n)) = \emptyset \\
 \mathcal{T}_{\text{Kont}}(\mathbf{letk}(x, e, \rho, a_k)) = \mathcal{T}_{\text{Env}}(\rho) \cup \{a_k\} & \mathcal{T}_{\text{Env}}(\rho) = \text{range}(\rho)
 \end{array}$$

Next, we introduce the adjacency relation between addresses, $(\rightsquigarrow_\varsigma) \subseteq \text{Loc} \times \text{Loc}$. Intuitively, $l \rightsquigarrow_\varsigma l'$ means that there is a reference from address l to l' , because the value (or continuation) stored in the store (or continuation store) at location l directly refers to location l' according to $\mathcal{T}_{\text{Val}}$ (or $\mathcal{T}_{\text{Kont}}$). We define the adjacency relation $(\rightsquigarrow_\varsigma)$ and the auxiliary adjacency relations $(\rightsquigarrow_{\sigma_\varsigma})$ and $(\rightsquigarrow_{\sigma_{k_\varsigma}})$:

$$\begin{array}{c}
 (\rightsquigarrow_\varsigma) = (\rightsquigarrow_{\sigma_\varsigma}) \cup (\rightsquigarrow_{\sigma_{k_\varsigma}}) \\
 a \rightsquigarrow_\sigma l \iff l \in \mathcal{T}_{\text{Val}}(\text{lookup}(\sigma, a)) \quad a_k \rightsquigarrow_{\sigma_k} l \iff l \in \mathcal{T}_{\text{Kont}}(\sigma_k(a_k))
 \end{array}$$

The auxiliary function $\mathcal{R}_{(\rightsquigarrow)} : \mathcal{P}(\text{Loc}) \rightarrow \mathcal{P}(\text{Loc})$ computes all addresses transitively reachable from a given root set using a transition relation $(\rightsquigarrow)$:

$$\mathcal{R}_{(\rightsquigarrow)}(\text{roots}) = \text{lfp}(f) \text{ \textbf{where} } f(S) = \text{roots} \cup \{l' \in \text{Loc} \mid \exists l \in S : l \rightsquigarrow l'\}$$

The first step of the tracing garbage collector is to determine all (transitively) reachable addresses, which could still be needed, from a given state ς . The function $\mathcal{R} : \Sigma \rightarrow \mathcal{P}(\text{Loc})$ computes all reachable addresses of a state:

$$\mathcal{R}(\varsigma) = \mathcal{R}_{(\rightsquigarrow_\varsigma)}(\mathcal{T}_\Sigma(\varsigma))$$

The next step is then to restrict the stores σ_ς and σ_{k_ς} to only contain the addresses in $\mathcal{R}(\varsigma)$. Therefore, the function $\Gamma : \Sigma \rightarrow \Sigma$ applies garbage collection to a state by restricting the domain of the store and the continuation store to $\mathcal{R}(\varsigma)$.

$$\Gamma(\varsigma) = \langle c_\varsigma, \sigma_\varsigma |_{\mathcal{R}(\varsigma)}, \sigma_{k_\varsigma} |_{\mathcal{R}(\varsigma)}, a_{k_\varsigma}, n_\varsigma \rangle$$

where $f|_X(x) = f(x)$ for $x \in X$ and $f|_X(x)$ is undefined for $x \notin X$.

In principle, the abstract machine can apply garbage collection at any arbitrary point throughout the evaluation. To keep the transition relation simple and deterministic, we can choose to apply garbage collection after every evaluation step. That is, we can define a new transition relation $(\rightarrow_\Gamma)$ as:

$$(\rightarrow_\Gamma) = \Gamma \circ (\rightarrow)$$

This means that $(\rightarrow_\Gamma)$ first takes an evaluation step, followed by a garbage collection. This “GC at every step” strategy makes the interpreter *garbage-free*: during evaluation, for every state ς , we have that $\mathcal{R}(\varsigma) = \text{dom}(\sigma_\varsigma) \cup \text{dom}(\sigma_{k_\varsigma})$. For a concrete interpreter, this property does not impact the result of the evaluation. Theorem 1 states this formally: for a given state ς , taking n transition steps using $(\rightarrow_\Gamma)$ leads to the same state, *modulo GC*, as taking n transition steps using $(\rightarrow)$.

Theorem 1 *If $\varsigma_0 \xrightarrow{n}_\Gamma \varsigma_1$ and $\varsigma_0 \xrightarrow{n} \varsigma_2$, then $\varsigma_1 = \Gamma(\varsigma_2)$.*

Proof. This follows directly from Theorem 6.10 of [106]: both $(\rightarrow)$ and $(\rightarrow_\Gamma)$ are subsets of the non-deterministic transition relation $(\Rightarrow)$ (defined in [106]). Hence, $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$ and by Lemma 6.8 of [106] and the definition of $(\rightarrow_\Gamma)$, $\Gamma(\varsigma_1) = \varsigma_1$, so that $\varsigma_1 = \Gamma(\varsigma_2)$. Proofs of Theorem 6.10 and Lemma 6.8 in [106] are given for a simple CPS-language, but can be repeated *mutatis mutandis* for λ_{ANF} . $\square$

Example. We revisit the running example of Figure 2.4, this time using the concrete semantics with garbage collection (i.e., using the transition relation $(\rightarrow_\Gamma)$). Theorem 1 guarantees that this results in the same sequence of states as the evaluation using $(\rightarrow)$, except that each state ς_i is replaced by $\varsigma_i^\Gamma = \Gamma(\varsigma_i)$.

$$\begin{aligned} \text{inject}(e_A) &= \varsigma_0^\Gamma = \langle \mathbf{ev}(e_A, []), [], [], a_{\text{halt}}, 0 \rangle \\ &\vdots \\ \rightarrow_\Gamma \varsigma_6^\Gamma &= \langle \mathbf{ev}(x, [x \mapsto 3]), [1 \mapsto v_{id}, 3 \mapsto v_f], [2 \mapsto \kappa_1], 2, 4 \rangle \quad [\text{E-APP}] \\ \rightarrow_\Gamma \varsigma_7^\Gamma &= \langle \mathbf{val}(v_f), [1 \mapsto v_{id}], [2 \mapsto \kappa_1], 2, 4 \rangle \quad [\text{E-ATOM}] \\ &\vdots \\ \rightarrow_\Gamma \varsigma_{16}^\Gamma &= \langle \mathbf{val}(\mathbf{num}(20)), [], [], a_{\text{halt}}, 9 \rangle \quad [\text{E-BINOP}] \end{aligned}$$

where v_{id} , v_f and κ_1 are as defined before.

This example highlights two places where garbage collection reduces the size of the value store and the continuation store. First, the continuation store in ς_6^Γ and ς_7^Γ no longer contains the old, unreachable continuation κ_0 (in fact, when using $\rightarrow_\Gamma$ each continuation is directly collected after it has been applied by the E-CONTINUE rule). Second, the value store in ς_7^Γ no longer contains the binding for the parameter x , as its address is no longer reachable from the current state (i.e., $\mathcal{T}_\Sigma(\varsigma_7) = \{1, 2\}$ and $\mathcal{R}(\varsigma_7) = \{1, 2, a_{\text{halt}}\}$). For the final state ς_{16}^Γ , both the value store and the continuation store are empty (as $\mathcal{R}(\varsigma_{16}) = \{a_{\text{halt}}\}$).

2.3 Abstract Semantics of λ_{ANF}

To reason about all possible run-time behaviours of a given program e , ideally an analysis would compute the set $S(e)$ of all program states that are reachable during the execution of e . However, $S(e)$ may be infinite for a program that does not terminate, or for a program where nondeterminism introduces an infinite number of possible execution paths. Therefore, such an analysis would be undecidable.

Rice’s theorem [134] confirms that it is impossible to build a decidable analysis for any non-trivial property about a program’s run-time behaviour, as captured by $S(e)$. The solution is to approximate (i.e., *abstract*) the behaviour of the program under analysis by interpreting it using the *abstract* language semantics. This approximation can be done in such a way that the resulting *abstract interpreter* [33] always terminates, resulting in a decidable analysis (i.e., an analysis that terminates for any given program). In general, the abstract language semantics aggregates a possibly infinite number of (possibly infinite) behaviours of a given program into a finite model capturing all its possible behaviours.

The cost of this approximation is a loss of precision. Whereas a concrete interpreter computes the exact behaviour of a given program², an abstract interpreter may be imprecise in two ways: either it includes behaviours (and results) which can never be encountered in any execution of the program (i.e., *false positives*) or it misses behaviours that can be encountered during some execution of the program (i.e., *false negatives*). The former arises from the abstract interpreter *over-approximating* its concrete counterpart, whereas the latter is a result of *under-approximation*. We can design the abstract interpreter to only over-approximate the behaviour of a concrete interpreter, guaranteeing that it never misses any behaviour that could be observed during a concrete execution of the program. In this case, the only imprecision comes from false

²Or multiple behaviours if the program contains nondeterminism.

positives. The resulting analysis is called *sound*³. In practice, soundness is a necessity for the analysis to prove important program properties, such as the absence of bugs (since the absence of bugs in the over-approximated program behaviour guarantees the absence of bugs for any real execution of that program).

Specifically, we follow the Abstracting Abstract Machines (AAM) approach [168, 105] to systematically derive an “abstract” version of the “concrete” abstract machine of Section 2.2. The key idea of AAM is that we render the state space Σ explored by the abstract interpreter finite so that $S(e)$ – a subset of Σ – is always guaranteed to be a finite (and thus computable) set for any program e . We describe the design of such an “abstract” abstract machine for λ_{ANF} ⁴.

State Space. The main issue with the concrete interpreter in Section 2.2 is that the state space Σ is infinite, and hence the set $S(e)$ is not always computable for any program e . To solve this issue, the AAM approach limits the infinite sets **Addr** and **KAddr** of the concrete interpreter to finite sets $\widehat{\text{Addr}}$ and $\widehat{\text{KAddr}}$, respectively. One can easily verify that this suffices to keep the state space finite. Figure 2.5 shows the definition⁵ for this *abstract* state space $\widehat{\Sigma}$.

$$\begin{array}{ll}
 \hat{\varsigma} \in \widehat{\Sigma} ::= \langle \hat{c}, \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle & \hat{v} \in \widehat{\text{Val}} = \widehat{\text{Clo}} \times \widehat{\text{Num}} \\
 \hat{c} \in \widehat{\text{Control}} ::= \text{ev}(e, \hat{\rho}, \hat{t}) & \hat{f} \in \widehat{\text{Clo}} = \mathcal{P}(\widehat{\text{Lam}} \times \widehat{\text{Env}}) \\
 & | \text{call}(\text{lam}, \hat{\rho}, \hat{v}, \hat{t}) \\
 & | \text{val}(\hat{v}) \\
 \hat{\rho} \in \widehat{\text{Env}} = \text{Var} \rightarrow \widehat{\text{Addr}} & \hat{n} \in \widehat{\text{Num}} ::= \perp_{\widehat{\text{Num}}} \mid \text{num} \\
 \hat{\sigma} \in \widehat{\text{Store}} = \widehat{\text{Addr}} \rightarrow \widehat{\text{Val}} & \hat{l} \in \widehat{\text{Loc}} = \widehat{\text{Addr}} \cup \widehat{\text{KAddr}} \\
 \hat{\sigma}_k \in \widehat{\text{KStore}} = \widehat{\text{KAddr}} \rightarrow \widehat{\text{Kont}} & \hat{a} \in \widehat{\text{Addr}} \text{ (a finite set)} \\
 & \hat{a}_k \in \widehat{\text{KAddr}} \text{ (a finite set)} \\
 & \hat{t} \in \widehat{\text{Time}} \text{ (a finite set)} \\
 \hat{\kappa} \in \widehat{\text{Kont}} = \mathcal{P}(\text{Var} \times \text{Exp} \times \widehat{\text{Env}} \times \widehat{\text{Time}} \times \widehat{\text{KAddr}}) &
 \end{array}$$

³The opposite property, *completeness*, would imply an absence of false positives. However, we know that an analysis cannot be sound and complete at the same time due to Rice’s theorem.

⁴Whenever we extend or modify a previous definition, important changes relative to the previous definition are highlighted in *grey*.

⁵By convention, hats are used to differentiate concrete components (e.g., Σ) from their abstract counterparts (e.g., $\widehat{\Sigma}$).

Figure 2.5: State space of the abstract interpreter for λ_{ANF} .

Limiting $\widehat{\text{Addr}}$ and $\widehat{\text{KAddr}}$ to finite sets has several implications.

First, since addresses are now drawn from finite sets, different addresses in the concrete interpreter can now correspond to only a single address in the abstract interpreter. Therefore, multiple values (and continuations) are sometimes allocated at the same address in $\hat{\sigma}$ (and $\hat{\sigma}_k$). To soundly over-approximate the behaviour of the concrete interpreter, values (and continuations) stored at the same address need to be *joined* together in such a way that the resulting abstract value at that address over-approximates all of the joined values. Therefore, a single abstract value now needs to represent multiple concrete values, and as such is defined as a product of a *set* of abstract closures (each possibly representing multiple concrete closures) and some abstraction representing a *set* of numbers. Since there is still an infinite number of sets of numbers, we currently either abstract numbers using $\perp_{\widehat{\text{Num}}}$ (meaning this abstract value does not represent any number) or **num** (meaning this abstract value can represent any number); we revisit more interesting abstractions in Section 2.3.1. Similarly, the continuation store now maps to a *set* of abstract continuations, so that multiple continuations allocated at the same address can easily be joined together by taking the union of both sets.

In general, a *join operator* $\sqcup$ can then be defined for values and continuations:

$$(\hat{f}_1, \hat{n}_1) \sqcup (\hat{f}_2, \hat{n}_2) = (\hat{f}_1 \cup \hat{f}_2, \hat{n}_1 \sqcup \hat{n}_2) \qquad \hat{\kappa}_1 \sqcup \hat{\kappa}_2 = \hat{\kappa}_1 \cup \hat{\kappa}_2$$

where we currently define the join operator for $\widehat{\text{Num}}$ as follows:

$$\hat{n}_1 \sqcup \hat{n}_2 = \begin{cases} \perp_{\widehat{\text{Num}}} & \text{if } \hat{n}_1 = \hat{n}_2 = \perp_{\widehat{\text{Num}}} \\ \mathbf{num} & \text{otherwise} \end{cases}$$

That is, when one of both abstract numbers can represent any number (**num**), the result of their join must by over-approximation also represent any number. We join different values at the same address using set unions for their respective sets of abstract closures and using the join operator for $\widehat{\text{Num}}$ for abstract numbers. When no closures are stored at an address, the corresponding set of closures for the abstract value at that address is empty (and similarly, $\perp_{\widehat{\text{Num}}}$ is used when no numbers are stored at that address). Similar to how $\perp_{\widehat{\text{Num}}}$ represents an empty set of numbers, we can explicitly introduce a *bottom value* $\perp_{\widehat{\text{Val}}}$ (and bottom continuation $\perp_{\widehat{\text{Kont}}}$) to represent an “empty” value (and continuation). That is, the abstract value $\perp_{\widehat{\text{Val}}}$ does not over-approximate any concrete closures or numbers.

$$\perp_{\widehat{\text{Val}}} = (\emptyset, \perp_{\widehat{\text{Num}}}) \qquad \perp_{\widehat{\text{Kont}}} = \emptyset$$

The initial store $\perp_{\widehat{\text{Store}}}$ (and continuation store $\perp_{\widehat{\text{KStore}}}$) contains no values (or continuations) for every address. Therefore, they are defined as follows:

$$\perp_{\widehat{\text{Store}}}(\hat{a}) = \perp_{\widehat{\text{Val}}} \qquad \perp_{\widehat{\text{KStore}}}(\hat{a}_k) = \perp_{\widehat{\text{Kont}}}$$

Whenever we *extend* the store with a new binding (e.g., upon a new allocation), the new value needs to be joined with the existing value at that address. Similarly, we need to perform such a join when *updating* a binding (e.g., upon mutating a reference). The reason we are not allowed to just overwrite the value and instead need to perform such a *weak update* is that a single abstract address may over-approximate multiple corresponding concrete addresses. An update would overwrite only a single concrete address, so to preserve soundness a join is necessary to avoid potentially overwriting the abstracted values of other concrete addresses. The abstract helper functions $\widehat{\text{extend}}$ and $\widehat{\text{update}}$ are defined as follows:

$$\widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}) = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v}) = \hat{\sigma}[\hat{a} \mapsto \hat{\sigma}(\hat{a}) \sqcup \hat{v}]$$

Analogously, we define a helper function $\widehat{\text{extend}}_k$ for the continuation store.

$$\widehat{\text{extend}}_k(\hat{\sigma}_k, \hat{a}_k, \hat{\kappa}) = \hat{\sigma}_k[\hat{a}_k \mapsto \hat{\sigma}_k(\hat{a}_k) \cup \hat{\kappa}]$$

The definition of $\widehat{\text{lookup}}$ remains identical to that of its concrete counterpart:

$$\widehat{\text{lookup}}(\hat{\sigma}, \hat{a}) = \hat{\sigma}(\hat{a})$$

The second consideration of using only a finite set of addresses is deciding *when* addresses need to be reused. This is done through the allocation functions $\widehat{\text{alloc}}$ and $\widehat{\text{alloc}}_k$. Unlike their concrete counterparts, these functions are no longer injective; that is, they can return the same address for different allocations (which is necessary when using a finite set of addresses for a potentially infinite number of allocations). To allow for interesting implementations of these allocation functions, evaluation states now also include a *timestamp* $\hat{t} \in \widehat{\text{Time}}$. Intuitively, $\widehat{\text{Time}}$ represents some contextual information that can be included when allocating addresses. We then often choose $\widehat{\text{Addr}} = \widehat{\text{KAddr}} = \text{Exp} \times \widehat{\text{Time}}$, so that

different allocations at the same program location (identified using Exp) can be differentiated using some context (identified using $\widehat{Time}$). Using these definitions for $\widehat{Addr}$ and $\widehat{KAddr}$, we can implement the allocation functions as follows⁶:

$$\widehat{alloc}(x, \hat{t}) = (x, \hat{t}) \qquad \widehat{alloc}_k(e, \hat{t}) = (e, \hat{t})$$

The choice of $\widehat{Time}$ is arbitrary as long as it is a finite set, ensuring that the state space $\widehat{\Sigma}$ also remains finite. A function $\widehat{tick} : Exp \times \widehat{Time} \rightarrow \widehat{Time}$ should be provided to update the timestamp throughout the evaluation⁷. For instance, a common implementation is the following (for some $k \geq 0$):

$$\widehat{Time} = Exp^{\leq k} \qquad \widehat{tick}(e, \hat{t}) = [e : \hat{t}]_k$$

where $[s]_k$ takes the first k elements of a sequence s .

This results in a k -CFA style allocation policy [146], where the context corresponds to the last k call sites. Higher values of k therefore lead to richer contexts and larger sets for $\widehat{Addr}$ and $\widehat{KAddr}$. On the one hand, this positively impacts analysis precision, as potentially fewer addresses need to be reused. On the other hand, it may negatively impact analysis performance, as this also increases the size of the abstract state space $\widehat{\Sigma}$ that needs to be explored by the abstract interpreter.

In general, any choice of $\widehat{Time}$ and $\widehat{tick}$ yields a sound and decidable analysis (as long as $\widehat{Time}$ is a finite set). As such, it is usually left open as a parameter to configure analysis precision and performance, as it controls how often (as determined by the choice of $\widehat{Time}$) and when (as determined by $\widehat{tick}$) addresses need to be reused [168]. This choice determines the *context sensitivity* or *polyvariance* [55] of the abstract interpreter. In the special case where no context is used (i.e., when using $k = 0$ in a k -CFA allocation policy), the abstract interpreter (and the corresponding analysis) is considered *context-insensitive*.

Transition Relation. The atomic evaluation function $\widehat{\mathcal{A}} : AExp \times \widehat{Env} \times \widehat{Store} \rightarrow \widehat{Val}$ now evaluates atomic expressions to an abstract value. The only changes compared to the definition of $\mathcal{A}$ are due to the representation of abstract values.

⁶Note that the signatures of these functions have been modified compared to their concrete counterparts: they now expect a timestamp $\hat{t}$ instead of a state ζ as their second argument.

⁷We assume that the timestamp only needs to be updated when stepping into a function, passing the operator f and the current timestamp $\hat{t}$ as arguments. This signature can trivially be generalised to include other contextual arguments for more advanced implementations.

$\hat{\mathcal{A}}(x, \hat{\rho}, \hat{\sigma}) = \widehat{\text{lookup}}(\hat{\sigma}, \hat{\rho}(x))$	[A-VAR]
$\hat{\mathcal{A}}(\text{lam}, \hat{\rho}, \hat{\sigma}) = (\{(\text{lam}, \hat{\rho})\}, \perp_{\widehat{\text{Num}}})$	[A-LAM]
$\hat{\mathcal{A}}(n, \hat{\rho}, \hat{\sigma}) = (\emptyset, \mathbf{num})$	[A-NUM]

Figure 2.6 shows the abstract transition relation ($\hat{\Rightarrow}$) for λ_{ANF} .

Unlike in the concrete interpreter for λ_{ANF} , in the abstract interpreter ($\hat{\Rightarrow}$) is no longer deterministic, as it needs to over-approximate *all* possible program behaviours to obtain a sound analysis. For instance, since a single abstract value can represent multiple closures, rule E-CALL can now apply any of all the closures represented by that value, generating multiple successor states. Similarly, rule E-CONTINUE needs to take all possible continuations it can return to into account.

Another difference compared to ($\rightarrow$) is that the time component is passed throughout the evaluation and updated when entering a function body (in rule E-CALL). As previously mentioned, this timestamp is then used by the allocation functions whenever a new binding needs to be added to the store.⁸

The abstract function $\hat{\mathcal{I}}$ needs to be adapted to correctly interpret operations for abstract numerical values. Since numbers are currently represented as either **num** (representing any number) or $\perp_{\widehat{\text{Num}}}$ (representing no number), we can just define $\hat{\mathcal{I}} : Op \rightarrow \widehat{\text{Num}} \rightarrow \widehat{\text{Num}} \rightarrow \widehat{\text{Num}}$ as follows for any $op \in Op$:

$$\hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2) = \begin{cases} \mathbf{num} & \text{if } \hat{n}_1 = \hat{n}_2 = \mathbf{num} \\ \perp_{\widehat{\text{Num}}} & \text{otherwise} \end{cases}$$

That is, any operation involving two numbers is defined to yield another number (and $\perp_{\widehat{\text{Num}}}$ if either argument is not a number). More interesting definitions of $\hat{\mathcal{I}}$ are required when using other numerical abstract domains (cf. Section 2.3.1).

$$\frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle \mathbf{ev}(ae, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \hat{\Rightarrow} \langle \mathbf{val}(\hat{v}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle} \quad [\text{E-ATOM}]$$

⁸Note that we are not threading the time component throughout evaluation steps, but rather scope it to a function body. This usually corresponds to more natural formulations of context sensitivity compared to the threaded style found in the original AAM formulation [105]. E.g., when using call sites as contexts, this follows the formulation of m -CFA [109, 53] (the call sites of the m topmost stack frames) rather than that of k -CFA (those of the last k calls).

$$\begin{array}{c}
 \frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\sigma}' = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{ev}(\text{set } x := ae, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \rightrightarrows \langle \text{val}(\hat{v}), \hat{\sigma}', \hat{\sigma}_k, \hat{a}_k \rangle} \text{[E-SET]} \\
 \\
 \frac{\begin{array}{c} (\hat{f}, -) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \\ (\text{lam}, \hat{\rho}') \in \hat{f} \quad \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \end{array}}{\langle \text{ev}(f(ae), \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \rightrightarrows \langle \text{call}(\text{lam}, \hat{\rho}', \hat{v}, \hat{t}'), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle} \text{[E-CALL]} \\
 \\
 \frac{\hat{a} = \widehat{\text{alloc}}(x, \hat{t}) \quad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{a}] \quad \hat{\sigma}' = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{call}(\lambda x.e, \hat{\rho}, \hat{v}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \rightrightarrows \langle \text{ev}(e, \hat{\rho}', \hat{t}), \hat{\sigma}', \hat{\sigma}_k, \hat{a}_k \rangle} \text{[E-APP]} \\
 \\
 \frac{\begin{array}{c} (-, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (-, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\ \hat{v} = (\emptyset, \widehat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \end{array}}{\langle \text{ev}(op(ae_1, ae_2), \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \rightrightarrows \langle \text{val}(\hat{v}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle} \text{[E-BINOP]} \\
 \\
 \frac{\begin{array}{c} \hat{a}'_k = \widehat{\text{alloc}}_k(e_1, \hat{t}) \\ \hat{\kappa} = \{ (x, e_2, \hat{\rho}, \hat{t}, \hat{a}_k) \} \quad \hat{\sigma}'_k = \widehat{\text{extend}}_k(\hat{\sigma}_k, \hat{a}_k, \hat{\kappa}) \end{array}}{\langle \text{ev}(\text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \rightrightarrows \langle \text{ev}(e_1, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}'_k, \hat{a}'_k \rangle} \text{[E-LET]} \\
 \\
 \frac{\begin{array}{c} (x, e, \hat{\rho}, \hat{t}, \hat{a}'_k) \in \hat{\sigma}_k(\hat{a}_k) \quad \hat{a} = \widehat{\text{alloc}}(x, \hat{t}) \\ \hat{\rho}' = \hat{\rho}[x \mapsto \hat{a}] \quad \hat{\sigma}' = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}) \end{array}}{\langle \text{val}(\hat{v}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \rightrightarrows \langle \text{ev}(e, \hat{\rho}', \hat{t}), \hat{\sigma}', \hat{\sigma}_k, \hat{a}'_k \rangle} \text{[E-CONTINUE]}
 \end{array}$$

 Figure 2.6: Transition rules of the abstract interpreter for λ_{ANF} .

Program Semantics. To evaluate a program using the *abstract* abstract machine, we again first construct the initial state, now using the *abstract* injection function $\widehat{\text{inject}}$. The only difference with its concrete counterpart is that the initial store (resp. continuation store) is $\perp_{\widehat{\text{Store}}}$ (resp. $\perp_{\widehat{\text{KStore}}}$) instead of being empty. We assume that $\hat{a}_{\text{halt}}$ is a special address reserved in the set $\widehat{\text{KAddr}}$, and $\hat{t}_0 \in \widehat{\text{Time}}$ is the initial timestamp (e.g., the empty sequence $[]$ when $\widehat{\text{Time}} = \widehat{\text{Exp}}^k$).

$$\widehat{\text{inject}}(e) = \langle \text{ev}(e, [], \hat{t}_0), \perp_{\widehat{\text{Store}}}, \perp_{\widehat{\text{KStore}}}, \hat{a}_{\text{halt}} \rangle$$

To define an abstract version of the `eval` function, we have to take the nondeterminism of the transition relation ($\Rightarrow$) into account. Therefore, we must join the resulting values for every possible execution path explored by the abstract interpreter:

$$\widehat{\text{eval}}(e) = \bigsqcup \{ \hat{v} \mid \widehat{\text{inject}}(e) \xRightarrow{*} \langle \text{val}(\hat{v}), -, -, \hat{a}_{\text{halt}} \rangle \}$$

We define the *abstract collecting semantics* of a program using a function $\hat{S} : \text{Exp} \rightarrow \mathcal{P}(\hat{\Sigma})$, which computes all states reachable from the initial state of the program using the abstract transition relation ($\Rightarrow$) (with transitive closure ($\xRightarrow{*}$)).

$$\hat{S}(e) = \{ \hat{\varsigma} \in \hat{\Sigma} \mid \widehat{\text{inject}}(e) \xRightarrow{*} \hat{\varsigma} \}$$

As the state space $\hat{\Sigma}$ of the abstract interpreter is finite, the important difference with the concrete collecting semantics is that $\hat{S}(e) \subseteq \hat{\Sigma}$ is also finite and therefore always computable for any given program e . Due to the over-approximating nature of ($\Rightarrow$), this set is an over-approximation of the reachable states for *all* possible executions of the program. Therefore, we can reason about the behaviour of e by reasoning about $\hat{S}(e)$ to obtain a sound and decidable program analysis. The cost of doing so is the precision loss resulting from the over-approximation (i.e., false positives due to states in $\hat{S}(e)$ that represent behaviours not possible in any concrete execution of the program — i.e., with no counterpart in $S(e)$).

2.3.1 Abstract Domains

The numerical abstraction provided by $\widehat{\text{Num}}$ ensures that the state space $\hat{\Sigma}$ remains finite, but is otherwise too imprecise to provide much interesting information about the program's numerical data flow. However, other non-finite abstractions are also possible under certain conditions.

First, any such abstract domain D (both for $\widehat{\text{Num}}$ as well as the overall abstract domain $\widehat{\text{Val}}$) should form a *join-semilattice*: it should have a partial order ($\sqsubseteq$), a least element ($\perp$) and a join operator ($\sqcup$). Intuitively, $d_1 \sqsubseteq d_2$ implies that d_2 is a less (or equally) precise approximation compared to d_1 , in the sense that d_2 also over-approximates everything over-approximated by d_1 (i.e., d_2 *subsumes* d_1). The least element $\perp$ is subsumed by all other elements: $\forall d \in D : \perp \sqsubseteq d$. The join operator ($\sqcup$) : $D \rightarrow D \rightarrow D$ should merge two lattice elements into their least upper bound, so that $d_1 \sqcup d_2$ is the smallest element which over-approximates both d_1 and d_2 (i.e., $\forall d_1, d_2 \in D : d_1 \sqsubseteq d_1 \sqcup d_2 \wedge d_2 \sqsubseteq d_1 \sqcup d_2$, and no smaller element that subsumes both d_1 and d_2 exists). Some abstract domains also have a

greatest element $\top$ (referred to as *top*), so that $\forall d \in D : d \sqsubseteq \top$ (i.e., $\top$ is the least precise element). When working with lattices, we assume that the definitions of $(\perp)_X$, $(\sqcup)_X$, $(\sqsubseteq)_X$ and $(\top)_X$ for a type X^9 are lifted to functions, sets and tuples:

$\perp_{\mathcal{P}(A)} = \emptyset$	$\perp_{A \rightarrow B}(a) = \perp_B$
$s_1 \sqcup_{\mathcal{P}(A)} s_2 = s_1 \cup s_2$	$(f \sqcup_{A \rightarrow B} g)(a) = f(a) \sqcup_B g(a)$
$s_1 \sqsubseteq_{\mathcal{P}(A)} s_2 \iff s_1 \subseteq s_2$	$f \sqsubseteq_{A \rightarrow B} g \iff \forall a : f(a) \sqsubseteq_B g(a)$
$\top_{\mathcal{P}(A)} = A$	$\top_{A \rightarrow B}(a) = \top_B$
$\perp_{A \times B} = (\perp_A, \perp_B)$	
$(a_1, b_1) \sqcup_{A \times B} (a_2, b_2) = (a_1 \sqcup_A a_2, b_1 \sqcup_B b_2)$	
$(a_1, b_1) \sqsubseteq_{A \times B} (a_2, b_2) \iff a_1 \sqsubseteq_A a_2 \wedge b_1 \sqsubseteq_B b_2$	
$\top_{A \times B} = (\top_A, \top_B)$	

Note that not only $\widehat{\text{Num}}$, but also the overall abstract domain $\widehat{\text{Val}}$ forms a lattice (because $\widehat{\text{Clo}}$ is a powerset lattice, and because $\widehat{\text{Val}}$ itself is a tuple lattice). Similarly, $\widehat{\text{Kont}}$ (a powerset) and both $\widehat{\text{Store}}$ and $\widehat{\text{KStore}}$ (functions) form lattices.

Second, for the abstract interpreters considered so far (where stores only grow monotonically through joins), the abstract domain does not have to be finite: it suffices to adhere to the ascending chain condition (ACC). This condition states that every weakly ascending sequence of lattice values (i.e., a sequence $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \dots$) eventually converges to a stable lattice value (i.e., for every such sequence there exists some n so that $d_n = d_{n+1} = d_{n+2} = \dots$)¹⁰. The ascending chain condition is important to guarantee termination for the abstract interpreter. That is, when using an infinite abstract domain $\widehat{\text{Val}}$, both $\widehat{\text{Store}}$ and therefore also the overall state space $\hat{\Sigma}$ become infinite. However, when $\widehat{\text{Val}}$ adheres to the ascending chain condition, we can still ensure that $\hat{S}(e)$ remains finite for every program e , as for every transition $\hat{\zeta} \xrightarrow{*} \hat{\zeta}'$, we have that $\hat{\sigma}_{\hat{\zeta}} \sqsubseteq \hat{\sigma}_{\hat{\zeta}'}$ (since the store can only change using $\widehat{\text{extend}}$ or $\widehat{\text{update}}$, both of which perform a join in the store). Therefore, when taking a sequence of steps using $(\xrightarrow{*})$, the ascending chain condition ensures that the store eventually converges, which in turn ensures that $\hat{S}(e)$ remains finite. As we discuss in Sections 2.3.2 and 2.3.3, the ACC no longer suffices once abstract GC or abstract counting is introduced; we refer to Table 6.1 in Chapter 6 for an overview of the requirements on the abstract domain for each abstract interpreter considered in this dissertation.

⁹In what follows, we omit the subscript for X when the type can be inferred from the context.

¹⁰Note that every finite domain trivially satisfies the ascending chain condition.

Third, the abstract domain and its abstract operations need to correctly over-approximate the concrete domain and its concrete operations. Formally, for an abstract domain A and a concrete domain C , we assume a *Galois connection* between $(A, \sqsubseteq)$ and $(\mathcal{P}(C), \subseteq)$ (written as $(\mathcal{P}(C), \subseteq) \xleftrightarrow[\alpha]{\gamma} (A, \sqsubseteq)$).¹¹ It includes two monotone functions $\alpha : \mathcal{P}(C) \rightarrow A$ and $\gamma : A \rightarrow \mathcal{P}(C)$, which define the mapping between the concrete and abstract domain. The *abstraction* function α over-approximates a set of concrete elements using an element from the abstract domain; inversely, the *concretisation* function γ returns the set of concrete elements approximated by a given abstract value. By definition, for a Galois connection we require that $\forall c \in \mathcal{P}(C) : c \subseteq \gamma(\alpha(c))$ and that $\forall a \in A : \alpha(\gamma(a)) \sqsubseteq a$.¹² The first conjunct ensures that $\alpha(c)$ is a sound over-approximation of c (possibly losing precision, as $\gamma(\alpha(c))$ may be a strict superset of c). The second conjunct (together with the monotonicity requirement for α) ensures that $\alpha(c)$ is the most precise abstract element that over-approximates c .¹³ For operations on the abstract domain, we require that they soundly over-approximate their concrete counterpart. In the case of a numerical abstract domain $\widehat{\text{Num}}$ (and a corresponding Galois connection $(\mathcal{P}(\mathbb{N}), \subseteq) \xleftrightarrow[\alpha]{\gamma} (\widehat{\text{Num}}, \sqsubseteq_{\widehat{\text{Num}}})$), we require that:

$$\forall op, \hat{n}_1, \hat{n}_2 : \forall n_1 \in \gamma(\hat{n}_1) : \forall n_2 \in \gamma(\hat{n}_2) : \mathcal{I}(op, n_1, n_2) \in \gamma(\hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2))$$

This property states that for every operation, the result of an abstract interpretation of that operation (on the corresponding abstracted arguments) over-approximates the result of a concrete interpretation of that operation. This requirement ensures that $\hat{\mathcal{I}}$ preserves the soundness of the abstract interpreter.

In addition, values should be abstracted soundly: whenever the abstract interpreter represents a set of concrete values c using an abstract value a , we require that $c \subseteq \gamma(a)$. For a numerical abstract domain $\widehat{\text{Num}}$, we therefore require that a literal number n is evaluated (in rule A-NUM) to an abstract value $\hat{n} \in \widehat{\text{Num}}$ such that $n \in \gamma(\hat{n})$ (the most precise abstraction being $\alpha(\{n\})$).

Choosing an appropriate abstract domain usually requires a balancing act between the precision and the performance (i.e., complexity) of the abstract

¹¹We could weaken this assumption, as our analyses do not make use of α . Therefore, strictly speaking, a Galois connection is not *required*, and we only need an abstract domain with a monotone concretisation function γ (as in [34]). This relaxation would admit abstract domains where no most precise over-approximation exists (e.g., polyhedra domains [36]).

¹²Usually, the defining requirement of a Galois connection is given more succinctly with the following (equivalent) property: $\forall a \in A, \forall c \in \mathcal{P}(C) : \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$.

¹³In fact, for a static analysis, we usually even have a *Galois insertion*, a special case of a Galois connection where $\forall a \in A : \alpha(\gamma(a)) = a$, written as $(\mathcal{P}(C), \subseteq) \xleftrightarrow[\alpha]{\gamma} (A, \sqsubseteq)$. This occurs in Galois connections where α is surjective (or, equivalently, where γ is injective).

interpreter. A larger abstract domain can in general offer increased precision: more abstract values can represent different sets of concrete values, potentially allowing for a more specific (i.e., less over-approximating) abstraction for a given concrete value. However, this can come at a cost of performance: a larger domain $\widehat{\text{Val}}$ leads to a larger state space $\widehat{\Sigma}$ to explore; in particular, the lattice “height” may determine how quickly values converge after successive updates in the store.

Constant Propagation Domain. A simple yet practical abstraction for a concrete domain C is a *constant propagation domain* (denoted as $\mathbf{CP}_C$). A constant propagation domain preserves full precision for abstract values that represent a single known (i.e., constant) value from the concrete domain C , and does not preserve any precision when they do not represent a known constant. Formally, we can define a lattice for $\mathbf{CP}_C$ and a corresponding Galois connection $(\mathcal{P}(C), \subseteq) \xleftrightarrow[\alpha]{\gamma} (\mathbf{CP}_C, \sqsubseteq)$.

$$\begin{array}{l}
 c \in C \text{ (a concrete domain)} \\
 \hat{v} \in \mathbf{CP}_C ::= \perp_{\mathbf{CP}_C} \\
 \quad | \text{cst}(c) \\
 \quad | \top_{\mathbf{CP}_C}
 \end{array}
 \quad
 \hat{v}_1 \sqcup \hat{v}_2 = \begin{cases} \hat{v}_2 & \text{if } \hat{v}_1 = \perp_{\mathbf{CP}_C} \\ \hat{v}_1 & \text{if } \hat{v}_2 = \perp_{\mathbf{CP}_C} \\ \hat{v}_1 & \text{if } \hat{v}_1 = \hat{v}_2 \\ \top_{\mathbf{CP}_C} & \text{otherwise} \end{cases}$$

$$\forall c \in C : \perp_{\mathbf{CP}_C} \sqsubseteq \text{cst}(c) \sqsubseteq \top_{\mathbf{CP}_C}$$

$$\alpha(S) = \begin{cases} \perp_{\mathbf{CP}_C} & \text{if } S = \emptyset \\ \text{cst}(c) & \text{if } S = \{c\} \\ \top_{\mathbf{CP}_C} & \text{otherwise} \end{cases}
 \quad
 \gamma(\hat{v}) = \begin{cases} \emptyset & \text{if } \hat{v} = \perp_{\mathbf{CP}_C} \\ \{c\} & \text{if } \hat{v} = \text{cst}(c) \\ C & \text{otherwise} \end{cases}$$

For instance, we can use $\mathbf{CP}_{\text{Num}}$ for the abstract numerical domain $\widehat{\text{Num}}$. The abstract interpretation function $\hat{\mathcal{I}}$ can then be defined soundly as follows:

$$\hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2) = \begin{cases} \perp_{\mathbf{CP}_{\text{Num}}} & \text{if } \hat{n}_1 = \perp_{\mathbf{CP}_{\text{Num}}} \vee \hat{n}_2 = \perp_{\mathbf{CP}_{\text{Num}}} \\ \text{cst}(\mathcal{I}(op, n_1, n_2)) & \text{if } \hat{n}_1 = \text{cst}(n_1) \wedge \hat{n}_2 = \text{cst}(n_2) \\ \top_{\mathbf{CP}_{\text{Num}}} & \text{otherwise} \end{cases}$$

Numbers can be abstracted precisely, i.e., rule A-NUM is adapted as follows:

$$\widehat{\mathcal{A}}(n, \hat{\rho}, \hat{\sigma}) = (\emptyset, \alpha(\{n\})) = (\emptyset, \mathbf{cst}(n)) \quad [\text{A-Num}]$$

Figure 2.7 shows the constant propagation lattice $\mathbf{CP}_{\text{Num}}$ more concisely using a Hasse diagram, which depicts all lattice elements and the subsumption relation between them. An upward edge from x to y implies that $x \sqsubseteq y$ ¹⁴ (consequently, we lose precision when “going up” in the diagram by increasing over-approximation).

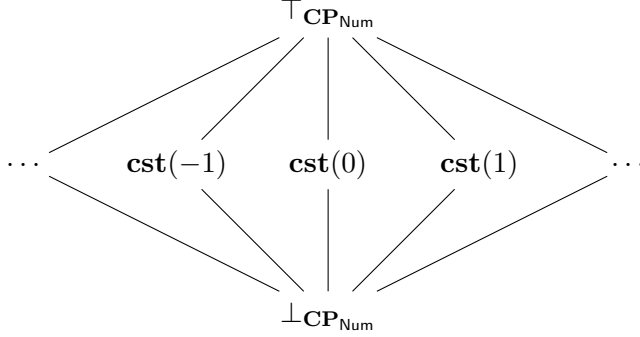


Figure 2.7: Hasse diagram of the constant propagation lattice $\mathbf{CP}_{\text{Num}}$.

Note that the constant propagation lattice $\mathbf{CP}_{\text{Num}}$ has an infinite number of elements (because the lattice has an infinite “width”), but adheres to the ascending chain condition (because the lattice has a finite “height”).

Example. We illustrate how the program e_A (Figure 2.4) is evaluated using the abstract semantics (i.e., using the transition relation $(\rightrightarrows)$) with $\mathbf{CP}_{\text{Num}}$ as the abstract numerical domain. The interpreter uses no context sensitivity, and therefore $\widehat{\text{Time}} = \{\epsilon\}$. We write $\mathbb{Q}x$ for the address of a variable x , i.e., $\mathbb{Q}x = \text{alloc}(x, \epsilon) = (x, \epsilon)$; analogously, for every expression $\text{let } x = e_1 \text{ in } e_2$, the continuation address is allocated at $\mathbb{Q}_k x = \text{alloc}_k(e_1, \epsilon) = (e_1, \epsilon)$. Finally, $[\hat{a}_1 \mapsto \hat{v}_1, \dots, \hat{a}_n \mapsto \hat{v}_n]$ denotes the store $\perp_{\widehat{\text{Store}}}[\hat{a}_1 \mapsto \hat{v}_1] \dots [\hat{a}_n \mapsto \hat{v}_n]$ (so that $\square$ denotes $\perp_{\widehat{\text{Store}}}$), and analogously for continuation stores. The state exploration is shown below:

$$\begin{aligned} \widehat{\text{inject}}(e_A) &= \hat{\varsigma}_0 = \langle \mathbf{ev}(e_A, \square, \epsilon), \square, \square, \hat{a}_{\text{halt}} \rangle \\ &\rightrightarrows \hat{\varsigma}_1 = \langle \mathbf{ev}(\lambda x.x, \square, \epsilon), \square, \hat{\sigma}_k^1, \mathbb{Q}_k \text{id} \rangle & [\text{E-LET}] \\ &\rightrightarrows \hat{\varsigma}_2 = \langle \mathbf{val}(\hat{v}_{\text{id}}), \square, \hat{\sigma}_k^1, \mathbb{Q}_k \text{id} \rangle & [\text{E-ATOM}] \\ &\rightrightarrows \hat{\varsigma}_3 = \langle \mathbf{ev}(e_B, \hat{\rho}_{\text{id}}, \epsilon), \hat{\sigma}^1, \hat{\sigma}_k^1, \hat{a}_{\text{halt}} \rangle & [\text{E-CONTINUE}] \\ &\rightrightarrows \hat{\varsigma}_4 = \langle \mathbf{ev}(\text{id}(\lambda a.\text{plus}(a, 1)), \hat{\rho}_{\text{id}}, \epsilon), \hat{\sigma}^1, \hat{\sigma}_k^2, \mathbb{Q}_k f \rangle & [\text{E-LET}] \end{aligned}$$

¹⁴We define a strict (non-reflexive) version of $\sqsubseteq$ as $d_1 \sqsubset d_2 \iff d_1 \sqsubseteq d_2 \wedge d_1 \neq d_2$.

$$\begin{array}{ll}
 \hat{\Rightarrow} \hat{\varsigma}_5 = \langle \mathbf{call}(\lambda x.x, [], \hat{v}_f, \epsilon), \hat{\sigma}^1, \hat{\sigma}_k^2, \mathbb{Q}_k f \rangle & [\text{E-CALL}] \\
 \hat{\Rightarrow} \hat{\varsigma}_6 = \langle \mathbf{ev}(x, \hat{\rho}_x, \epsilon), \hat{\sigma}^2, \hat{\sigma}_k^2, \mathbb{Q}_k f \rangle & [\text{E-APP}] \\
 \hat{\Rightarrow} \hat{\varsigma}_7 = \langle \mathbf{val}(\hat{v}_f), \hat{\sigma}^2, \hat{\sigma}_k^2, \mathbb{Q}_k f \rangle & [\text{E-ATOM}] \\
 \hat{\Rightarrow} \hat{\varsigma}_8 = \langle \mathbf{ev}(e_C, \hat{\rho}_{id,f}, \epsilon), \hat{\sigma}^3, \hat{\sigma}_k^2, \hat{a}_{\text{halt}} \rangle & [\text{E-CONTINUE}] \\
 \hat{\Rightarrow} \hat{\varsigma}_9 = \langle \mathbf{ev}(id(\lambda b.plus(b, b)), \hat{\rho}_{id,f}, \epsilon), \hat{\sigma}^3, \hat{\sigma}_k^3, \mathbb{Q}_k g \rangle & [\text{E-LET}] \\
 \hat{\Rightarrow} \hat{\varsigma}_{10} = \langle \mathbf{call}(\lambda x.x, [], \hat{v}_g, \epsilon), \hat{\sigma}^3, \hat{\sigma}_k^3, \mathbb{Q}_k g \rangle & [\text{E-CALL}] \\
 \hat{\Rightarrow} \hat{\varsigma}_{11} = \langle \mathbf{ev}(x, \hat{\rho}_x, \epsilon), \hat{\sigma}^4, \hat{\sigma}_k^3, \mathbb{Q}_k g \rangle & [\text{E-APP}] \\
 \hat{\Rightarrow} \hat{\varsigma}_{12} = \langle \mathbf{val}(\hat{v}_f \sqcup \hat{v}_g), \hat{\sigma}^4, \hat{\sigma}_k^3, \mathbb{Q}_k g \rangle & [\text{E-ATOM}] \\
 \hat{\Rightarrow} \hat{\varsigma}_{13} = \langle \mathbf{ev}(e_D, \hat{\rho}_{id,f,g}, \epsilon), \hat{\sigma}^5, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle & [\text{E-CONTINUE}] \\
 \hat{\varsigma}_{13} \hat{\Rightarrow} \hat{\varsigma}_{14} & \hat{\varsigma}_{13} \hat{\Rightarrow} \hat{\varsigma}'_{14} & [\text{E-CALL}] \\
 \hat{\Rightarrow} \hat{\varsigma}_{15} & \hat{\Rightarrow} \hat{\varsigma}'_{15} & [\text{E-APP}] \\
 \hat{\Rightarrow} \hat{\varsigma}_{16} & \hat{\Rightarrow} \hat{\varsigma}'_{16} & [\text{E-BINOP}]
 \end{array}$$

where

$$\begin{aligned}
 \hat{\varsigma}_{14} &= \langle \mathbf{call}(\lambda b.plus(b, b), \hat{\rho}_{id,f}, \hat{v}_{10}, \epsilon), \hat{\sigma}^5, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle \\
 \hat{\varsigma}_{15} &= \langle \mathbf{ev}(plus(b, b), \hat{\rho}_{id,f,b}, \epsilon), \hat{\sigma}^6, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle \\
 \hat{\varsigma}_{16} &= \langle \mathbf{val}(\hat{v}_{20}), \hat{\sigma}^6, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle \\
 \hat{\varsigma}'_{14} &= \langle \mathbf{call}(\lambda a.plus(a, 1), \hat{\rho}_{id}, \hat{v}_{10}, \epsilon), \hat{\sigma}^5, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle \\
 \hat{\varsigma}'_{15} &= \langle \mathbf{ev}(plus(a, 1), \hat{\rho}_{id,a}, \epsilon), \hat{\sigma}^7, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle \\
 \hat{\varsigma}'_{16} &= \langle \mathbf{val}(\hat{v}_{11}), \hat{\sigma}^7, \hat{\sigma}_k^3, \hat{a}_{\text{halt}} \rangle
 \end{aligned}$$

$$\hat{\rho}_{x_1, \dots, x_n} = [x_1 \mapsto \mathbb{Q}x_1, \dots, x_n \mapsto \mathbb{Q}x_n]$$

$$\begin{array}{ll}
 \hat{v}_{id} = (\{(\lambda x.x, [])\}, \perp_{\mathbf{CP}_{\text{Num}}}) & \hat{v}_f = (\{(\lambda a.plus(a, 1), \hat{\rho}_{id})\}, \perp_{\mathbf{CP}_{\text{Num}}}) \\
 \hat{v}_n = (\emptyset, \mathbf{cst}(n)) & \hat{v}_g = (\{(\lambda b.plus(b, b), \hat{\rho}_{id,f})\}, \perp_{\mathbf{CP}_{\text{Num}}}) \\
 \hat{\kappa}_0 = \{(id, e_B, [], \epsilon, \hat{a}_{\text{halt}})\} & \hat{\sigma}_k^1 = [\mathbb{Q}_k id \mapsto \hat{\kappa}_0] \\
 \hat{\kappa}_1 = \{(f, e_C, \hat{\rho}_{id}, \epsilon, \hat{a}_{\text{halt}})\} & \hat{\sigma}_k^2 = \hat{\sigma}_k^1[\mathbb{Q}_k f \mapsto \hat{\kappa}_1] \\
 \hat{\kappa}_2 = \{(g, e_D, \hat{\rho}_{id,f}, \epsilon, \hat{a}_{\text{halt}})\} & \hat{\sigma}_k^3 = \hat{\sigma}_k^2[\mathbb{Q}_k g \mapsto \hat{\kappa}_2] \\
 \hat{\sigma}^1 = [\mathbb{Q}id \mapsto \hat{v}_{id}] & \hat{\sigma}^5 = \hat{\sigma}^4[\mathbb{Q}g \mapsto \hat{v}_f \sqcup \hat{v}_g] \\
 \hat{\sigma}^2 = \hat{\sigma}^1[\mathbb{Q}x \mapsto \hat{v}_f] & \hat{\sigma}^6 = \hat{\sigma}^5[\mathbb{Q}b \mapsto \hat{v}_{10}] \\
 \hat{\sigma}^3 = \hat{\sigma}^2[\mathbb{Q}f \mapsto \hat{v}_f] & \hat{\sigma}^7 = \hat{\sigma}^5[\mathbb{Q}a \mapsto \hat{v}_{10}] \\
 \hat{\sigma}^4 = \hat{\sigma}^3[\mathbb{Q}x \mapsto \hat{v}_f \sqcup \hat{v}_g]
 \end{array}$$

Compared to the concrete interpretation of e_A (Section 2.2), the abstract interpreter loses precision in the transition $\hat{\varsigma}_{10} \hat{\Rightarrow} \hat{\varsigma}_{11}$. This precision loss is due to the reuse of the same address $\mathbb{Q}x$ for the second call to id , causing the abstract

interpreter to join the values $\hat{v}_f$ and $\hat{v}_g$ at that address (i.e., @x gets bound to $\hat{v}_f \sqcup \hat{v}_g$). As this imprecise abstract value propagates to the binding of g , the abstract interpreter at state $\hat{\varsigma}_{13}$ consequently does not know whether function value $\hat{v}_f$ or $\hat{v}_g$ is called. To soundly over-approximate all possible execution behaviours, the abstract interpreter considers both options, and therefore branches (as per the E-CALL rule) into two successor states $\hat{\varsigma}_{14}$ and $\hat{\varsigma}'_{14}$. The second branch corresponds to a spurious path, as it does not occur in a real execution of the program (which also harms the performance of the abstract interpreter, as time is wasted on the exploration of such spurious paths). Since both branches reach a final state, the result of the abstract interpretation is $\widehat{\text{eval}}(e_A) = \hat{v}_{11} \sqcup \hat{v}_{20} = (\emptyset, \top_{\text{CP}_{\text{Num}}})$, which over-approximates the concrete result $\text{eval}(e_A) = \text{num}(20)$. The abstract collecting semantics for this program is $\hat{S}(e_A) = \{\hat{\varsigma}_i\}_{i=0}^{16} \cup \{\hat{\varsigma}'_{14}, \hat{\varsigma}'_{15}, \hat{\varsigma}'_{16}\}$.

2.3.2 Abstract Garbage Collection

Just as in the concrete interpreter (Section 2.2.1), one can perform garbage collection in the abstract interpreter (known as *abstract garbage collection*) by removing all addresses from $\hat{\sigma}_{\hat{\varsigma}}$ and $\hat{\sigma}_{k\hat{\varsigma}}$ that are no longer reachable from $\hat{\varsigma}$.

We adapt the definitions of the auxiliary functions $\widehat{\mathcal{T}}_X$ to take into account the new representation of values and continuations. The addresses reachable from an abstract value (resp. continuation) are the union of all addresses reachable by all closures (resp. all continuation frames) it over-approximates.

$$\begin{aligned}
\widehat{\mathcal{T}}_{\hat{\Sigma}}(\langle \hat{c}, \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle) &= \widehat{\mathcal{T}}_{\text{Control}}(\hat{c}) \cup \{\hat{a}_k\} \\
\widehat{\mathcal{T}}_{\text{Control}}(\text{ev}(e, \hat{\rho}, \hat{t})) &= \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}) \\
\widehat{\mathcal{T}}_{\text{Control}}(\text{call}(\text{lam}, \hat{\rho}, \hat{v}, \hat{t})) &= \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}) \cup \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}) \\
\widehat{\mathcal{T}}_{\text{Control}}(\text{val}(\hat{v})) &= \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}) \\
\widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}) &= \text{range}(\hat{\rho}) \\
\widehat{\mathcal{T}}_{\text{Val}}((\hat{f}, \hat{n})) &= \widehat{\mathcal{T}}_{\text{Clo}}(\hat{f}) \\
\widehat{\mathcal{T}}_{\text{Clo}}(\hat{f}) &= \bigcup_{(\text{lam}, \hat{\rho}) \in \hat{f}} \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}) \\
\widehat{\mathcal{T}}_{\text{Kont}}(\hat{k}) &= \bigcup_{(x, e, \hat{\rho}, \hat{t}, \hat{a}_k) \in \hat{k}} \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}) \cup \{\hat{a}_k\}
\end{aligned}$$

The definitions of $(\rightsquigarrow_{\hat{\varsigma}})$, $\widehat{\mathcal{R}}$ and $\widehat{\Gamma}$ remain essentially unchanged.

$$\begin{aligned}
 \hat{a} \rightsquigarrow_{\hat{\sigma}} \hat{l} &\iff \hat{l} \in \widehat{\mathcal{T}}_{\widehat{\text{Val}}}(\widehat{\text{lookup}}(\hat{\sigma}, \hat{a})) & \hat{a}_k \rightsquigarrow_{\hat{\sigma}_k} \hat{l} &\iff \hat{l} \in \widehat{\mathcal{T}}_{\widehat{\text{Kont}}}(\hat{\sigma}_k(\hat{a}_k)) \\
 \widehat{\mathcal{R}}_{(\rightsquigarrow)}(\text{roots}) &= \text{lfp}(f) \text{ \textbf{where} } f(S) = \text{roots} \cup \{\hat{l}' \in \widehat{\text{Loc}} \mid \exists \hat{l} \in S : \hat{l} \rightsquigarrow \hat{l}'\} \\
 (\rightsquigarrow_{\hat{\zeta}}) &= (\rightsquigarrow_{\hat{\sigma}_{\hat{\zeta}}}) \cup (\rightsquigarrow_{\hat{\sigma}_{k\hat{\zeta}}}) & \widehat{\mathcal{R}}(\hat{\zeta}) &= \widehat{\mathcal{R}}_{(\rightsquigarrow_{\hat{\zeta}})}(\widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\zeta})) \\
 \widehat{\Gamma}(\hat{\zeta}) &= \langle c_{\hat{\zeta}}, \hat{\sigma}_{\hat{\zeta}}|_{\widehat{\mathcal{R}}(\hat{\zeta})}, \hat{\sigma}_{k\hat{\zeta}}|_{\widehat{\mathcal{R}}(\hat{\zeta})}, \hat{a}_{k\hat{\zeta}} \rangle
 \end{aligned}$$

where $\hat{\sigma}|_X(\hat{a}) = \hat{\sigma}(\hat{a})$ for $\hat{a} \in X$ and $\hat{\sigma}(\hat{a}) = \perp_{\widehat{\text{Val}}}$ for $\hat{a} \notin X$. Analogously, we define $\hat{\sigma}_k|_X(\hat{a}_k) = \hat{\sigma}_k(\hat{a}_k)$ for $\hat{a}_k \in X$ and $\hat{\sigma}_k|_X(\hat{a}_k) = \perp_{\widehat{\text{Kont}}}$ for $\hat{a}_k \notin X$.

We define the new transition relation $(\rightrightarrows_{\widehat{\Gamma}})$ as a composition of $\widehat{\Gamma}$ and $(\rightrightarrows)$:

$$(\rightrightarrows_{\widehat{\Gamma}}) = \widehat{\Gamma} \circ (\rightrightarrows)$$

Applying garbage collection at every evaluation step ensures that we end up with a *garbage-free* abstract interpreter, as no garbage survives in any state produced by $(\rightrightarrows_{\widehat{\Gamma}})$ for each evaluation step. More formally, we define $\widehat{\mathcal{S}} : \widehat{\Sigma} \rightarrow \mathcal{P}(\widehat{\text{Loc}})$, so that $\widehat{\mathcal{S}}(\hat{\zeta})$ is the set of addresses bound in the stores of $\hat{\zeta}$, as follows:

$$\widehat{\mathcal{S}}(\hat{\zeta}) = \{\hat{a} \in \widehat{\text{Addr}} \mid \hat{\sigma}_{\hat{\zeta}}(\hat{a}) \neq \perp_{\widehat{\text{Val}}}\} \cup \{\hat{a}_k \in \widehat{\text{KAddr}} \mid \hat{\sigma}_{k\hat{\zeta}}(\hat{a}_k) \neq \perp_{\widehat{\text{Kont}}}\}$$

For a garbage-free abstract interpreter, garbage collection should collect *all* addresses $\hat{l} \notin \widehat{\mathcal{R}}(\hat{\zeta})$, so that $\widehat{\mathcal{S}}(\hat{\zeta}) \subseteq \widehat{\mathcal{R}}(\hat{\zeta})$. Intuitively, this implies that the garbage collection never misses any garbage, only keeping non-garbage bindings (i.e., reachable bindings which may still be needed later on) in the store at all times.

Definition 1 (Garbage-free) A state $\hat{\zeta}$ is garbage-free iff $\widehat{\mathcal{S}}(\hat{\zeta}) \subseteq \widehat{\mathcal{R}}(\hat{\zeta})$. A transition relation $(\rightrightarrows)$ is garbage-free iff for every garbage-free state $\hat{\zeta}$ where $\hat{\zeta} \rightrightarrows \hat{\zeta}'$, $\hat{\zeta}'$ is garbage-free. That is, iff $(\rightrightarrows)$ preserves garbage-freeness.

Corollary 1 $\forall \hat{\zeta} \in \widehat{\Sigma} : \widehat{\Gamma}(\hat{\zeta})$ is a garbage-free state.

Proof. Note that $\widehat{\mathcal{R}}(\widehat{\Gamma}(\hat{\zeta})) = \widehat{\mathcal{R}}(\hat{\zeta})$, since garbage collection by definition preserves all reachable bindings. Since $\widehat{\Gamma}$ restricts the store $\hat{\sigma}$ (and continuation store $\hat{\sigma}_k$) to the set of reachable bindings $\widehat{\mathcal{R}}(\hat{\zeta})$, it is clear from the definition of $\widehat{\mathcal{S}}$ that $\widehat{\mathcal{S}}(\widehat{\Gamma}(\hat{\zeta})) \subseteq \widehat{\mathcal{R}}(\hat{\zeta})$, and therefore $\widehat{\mathcal{S}}(\widehat{\Gamma}(\hat{\zeta})) \subseteq \widehat{\mathcal{R}}(\widehat{\Gamma}(\hat{\zeta}))$ (i.e., $\widehat{\Gamma}(\hat{\zeta})$ is garbage-free). $\square$

Corollary 2 *A state $\hat{\varsigma}$ is garbage-free $\iff \hat{\Gamma}(\hat{\varsigma}) = \hat{\varsigma}$.*

Proof.

- ($\implies$): Note that $\hat{\sigma}|_A|_B = \hat{\sigma}|_A$ if $A \subseteq B$, and that $\hat{\sigma}_{\hat{\varsigma}}|_{\hat{\mathcal{S}}(\hat{\varsigma})} = \hat{\sigma}_{\hat{\varsigma}}$ by definition of $\hat{\mathcal{S}}$ (analogous for $\hat{\sigma}_{k\hat{\varsigma}}$). Since $\hat{\varsigma}$ is garbage-free, $\hat{\mathcal{S}}(\hat{\varsigma}) \subseteq \hat{\mathcal{R}}(\hat{\varsigma})$. It follows that $\hat{\sigma}_{\hat{\varsigma}}|_{\hat{\mathcal{R}}(\hat{\varsigma})} = \hat{\sigma}_{\hat{\varsigma}}|_{\hat{\mathcal{S}}(\hat{\varsigma})}|_{\hat{\mathcal{R}}(\hat{\varsigma})} = \hat{\sigma}_{\hat{\varsigma}}|_{\hat{\mathcal{S}}(\hat{\varsigma})} = \hat{\sigma}_{\hat{\varsigma}}$. By similar reasoning for $\hat{\sigma}_{k\hat{\varsigma}}$ and by the definition of $\hat{\Gamma}$, we have that $\hat{\Gamma}(\hat{\varsigma}) = \hat{\varsigma}$.
- ($\impliedby$): By Corollary 1, $\hat{\Gamma}(\hat{\varsigma})$ is a garbage-free state, and since $\hat{\Gamma}(\hat{\varsigma}) = \hat{\varsigma}$, it directly follows that $\hat{\varsigma}$ is also garbage-free.

□

Theorem 2 *$(\hat{\rightrightarrows}_{\hat{\Gamma}})$ is garbage-free.*

Proof. $(\hat{\rightrightarrows}_{\hat{\Gamma}})$ is trivially garbage-free by construction: it applies $\hat{\Gamma}$ to the state produced by $(\hat{\rightrightarrows})$, which by Corollary 1 always results in a garbage-free state. □

The dual property of garbage-freeness is GC soundness. In terms of abstract GC, soundness means that addresses that are reachable are not removed from the store [106]. That is, garbage collection should collect *only* unreachable addresses $\hat{l} \notin \hat{\mathcal{R}}(\hat{\varsigma})$, so that $\hat{\mathcal{R}}(\hat{\varsigma}) \subseteq \hat{\mathcal{S}}(\hat{\varsigma})$. Intuitively, this implies that the garbage collection is safe, and never removes a binding that is still needed.

Definition 2 (GC Soundness) *A state $\hat{\varsigma}$ is GC sound iff $\hat{\mathcal{R}}(\hat{\varsigma}) \subseteq \hat{\mathcal{S}}(\hat{\varsigma})$. A transition relation $(\hat{\rightrightarrows})$ is GC sound iff for every GC sound state $\hat{\varsigma}$ where $\hat{\varsigma} \hat{\rightrightarrows} \hat{\varsigma}'$, $\hat{\varsigma}'$ is GC sound. That is, iff $(\hat{\rightrightarrows})$ preserves GC soundness.*

Theorem 3 *$(\hat{\rightrightarrows}_{\hat{\Gamma}})$ is GC sound.*

Proof. A simple case analysis reveals that $(\hat{\rightrightarrows})$ is GC sound. The GC function $\hat{\Gamma}$ is also GC sound by construction, as it only removes addresses from $\hat{\mathcal{S}}(\hat{\varsigma})$ that are not in $\hat{\mathcal{R}}(\hat{\varsigma})$. Hence, $(\hat{\rightrightarrows}_{\hat{\Gamma}})$, the composition of $(\hat{\rightrightarrows})$ and $\hat{\Gamma}$, is also GC sound. □

Note that GC soundness is always implied for a sound analysis. When the abstract interpreter is both sound and garbage-free, we have that $\hat{\mathcal{R}}(\hat{\varsigma}) = \hat{\mathcal{S}}(\hat{\varsigma})$.

Impact on Precision. Intuitively, garbage-freeness ensures that there is no garbage in the store at every step of the evaluation. For a concrete interpreter, garbage-freeness is not a relevant property, as Theorem 1 confirms that concrete garbage collection does not influence the interpreter's results. However, for an

abstract interpreter, abstract garbage collection can greatly improve precision. Note that “*merging [i.e., join] is the enemy of precision*” [77]: a join (i.e., $\hat{v}_1 \sqcup \hat{v}_2$) needs to compute some over-approximation of both $\hat{v}_1$ and $\hat{v}_2$, and future uses of this joined value can no longer distinguish between $\hat{v}_1$ or $\hat{v}_2$ (needing to take both into account each time to maintain a sound over-approximation). Such a join happens whenever the abstract interpreter reuses (through $\widehat{\text{alloc}}$ or $\widehat{\text{alloc}}_k$) an already allocated address (i.e., an address $\hat{l} \in \widehat{\mathcal{S}}(\zeta)$) when adding a new binding to the store, as the old value at $\hat{l}$ is joined with the new one (see $\widehat{\text{extend}}$). In general, such joins in the store (and the resulting precision losses) are unavoidable¹⁵, as addresses need to be reused to soundly over-approximate a possibly infinite store using only a finite number of addresses. However, sometimes these joins are in principle unnecessary: when the address was unreachable but still in use (i.e., not in $\widehat{\mathcal{R}}(\zeta)$ but still in $\widehat{\mathcal{S}}(\zeta)$) before being reallocated, the join happens with an old value that can be considered “garbage”, therefore unnecessarily reducing precision. Abstract GC avoids this kind of precision loss by removing unreachable addresses from the store, therefore eliminating the need to perform a join with “garbage values” at these addresses. That is, by limiting $\widehat{\mathcal{S}}(\zeta)$ to $\widehat{\mathcal{R}}(\zeta)$, it ensures that such precision-decreasing joins happen only when actually necessary.

Impact on Performance. As evidenced by the example below, abstract GC often has a positive impact on performance by avoiding the exploration of spurious paths. In addition, no time is wasted on exploring states multiple times that are equivalent up to GC (i.e., for states $\hat{\zeta}_1$ and $\hat{\zeta}_2$ where $\hat{\zeta}_1 \neq \hat{\zeta}_2$ but $\widehat{\Gamma}(\hat{\zeta}_1) = \widehat{\Gamma}(\hat{\zeta}_2)$).

However, applying garbage collection at every step (as in $(\widehat{\rhd}_{\widehat{\Gamma}})$) may not be practical, since computing the set $\widehat{\mathcal{R}}(\zeta)$ at every evaluation step incurs its own significant performance overhead. Some abstract interpreters therefore opt to apply abstract GC less frequently to avoid this overhead. Note that, when abstract GC is applied too infrequently, the abstract interpreter may suffer garbage-induced precision and performance issues (the latter due to spurious paths); when applied too frequently, the overhead of abstract GC may outweigh its benefits.

Another important implication of using abstract GC is that the ACC for the abstract domain no longer suffices to guarantee termination for the abstract interpreter. The reason is that for a transition $\zeta \xrightarrow{\widehat{\Gamma}} \zeta'$, we can no longer guarantee that $\hat{\sigma}_{\zeta} \sqsubseteq \hat{\sigma}_{\zeta'}$, since the abstract GC that occurred as part of $(\widehat{\rhd}_{\widehat{\Gamma}})$ may have removed addresses from the store. That is, the stores of a sequence of states produced by $(\widehat{\rhd}_{\widehat{\Gamma}})$ no longer form an ascending chain so that consequently, the ascending chain condition is no longer applicable. Convergence therefore needs to rely on a finite state space to ensure that $\hat{S}(e)$ remains finite (and therefore computable) as well

¹⁵Confirming Rice’s theorem [134]: a decidable analysis cannot have perfect precision.

for every program e . This implies that the abstract domain $\widehat{\text{Val}}$ (and therefore the numerical domain $\widehat{\text{Num}}$) also needs to be finite. Consequently, the abstract interpreter is no longer guaranteed to terminate when using the infinite constant propagation domain $\mathbf{CP}_{\text{Num}}$.

Example. We illustrate the impact of abstract GC by showing the state exploration of the abstract interpreter for the program e_A (Figure 2.4) with abstract GC applied at every step (i.e., using the transition relation $(\rightrightarrows_{\widehat{\Gamma}})$). The interpreter again uses $\mathbf{CP}_{\text{Num}}$ as the abstract numerical domain and is context-insensitive.

$$\begin{aligned}
\widehat{\text{inject}}(e_A) &= \hat{\varsigma}_0^{\widehat{\Gamma}} = \langle \mathbf{ev}(e_A, [], \epsilon), [], [], \hat{a}_{\text{halt}} \rangle \\
&\vdots \\
\rightrightarrows_{\widehat{\Gamma}} \hat{\varsigma}_7^{\widehat{\Gamma}} &= \langle \mathbf{val}(\hat{v}_f), \hat{\sigma}^1, \hat{\sigma}_k^4, \mathbb{Q}_k f \rangle && [\text{E-ATOM}] \\
&\vdots \\
\hat{\varsigma}_{11}^{\widehat{\Gamma}} &= \langle \mathbf{ev}(x, \hat{\rho}_x, \epsilon), \hat{\sigma}^9, \hat{\sigma}_k^5, \mathbb{Q}_k g \rangle && [\text{E-APP}] \\
\rightrightarrows_{\widehat{\Gamma}} \hat{\varsigma}_{12}^{\widehat{\Gamma}} &= \langle \mathbf{val}(\hat{v}_g), \hat{\sigma}^8, \hat{\sigma}_k^5, \mathbb{Q}_k g \rangle && [\text{E-ATOM}] \\
\rightrightarrows_{\widehat{\Gamma}} \hat{\varsigma}_{13}^{\widehat{\Gamma}} &= \langle \mathbf{ev}(e_D, \hat{\rho}_{id,f,g}, \epsilon), \hat{\sigma}^{10}, [], \hat{a}_{\text{halt}} \rangle && [\text{E-CONTINUE}] \\
\rightrightarrows_{\widehat{\Gamma}} \hat{\varsigma}_{14}^{\widehat{\Gamma}} &= \langle \mathbf{call}(\lambda b. \text{plus}(b, b), \hat{\rho}_{id,f}, \hat{v}_{10}, \epsilon), \hat{\sigma}^8, [], \hat{a}_{\text{halt}} \rangle && [\text{E-CALL}] \\
\rightrightarrows_{\widehat{\Gamma}} \hat{\varsigma}_{15}^{\widehat{\Gamma}} &= \langle \mathbf{ev}(\text{plus}(b, b), \hat{\rho}_{id,f,b}, \epsilon), \hat{\sigma}^{11}, [], \hat{a}_{\text{halt}} \rangle && [\text{E-APP}] \\
\rightrightarrows_{\widehat{\Gamma}} \hat{\varsigma}_{16}^{\widehat{\Gamma}} &= \langle \mathbf{val}(\hat{v}_{20}), [], [], \hat{a}_{\text{halt}} \rangle && [\text{E-BINOP}]
\end{aligned}$$

$$\begin{aligned}
\text{where } \hat{\sigma}^8 &= \hat{\sigma}^1[\mathbb{Q}f \mapsto \hat{v}_f] && \hat{\sigma}^{10} = \hat{\sigma}^8[\mathbb{Q}g \mapsto \hat{v}_g] \\
\hat{\sigma}^9 &= \hat{\sigma}^8[\mathbb{Q}x \mapsto \hat{v}_g] && \hat{\sigma}^{11} = \hat{\sigma}^8[\mathbb{Q}b \mapsto \hat{v}_{10}] \\
\hat{\sigma}_k^4 &= [\mathbb{Q}_k f \mapsto \hat{\kappa}_1] && \hat{\sigma}_k^5 = [\mathbb{Q}_k g \mapsto \hat{\kappa}_2]
\end{aligned}$$

All other components are the same as defined before.

Without abstract GC (Section 2.3.1), the abstract interpreter loses precision because both calls to *id* reuse the same address $\mathbb{Q}x$ for their argument binding and the argument values of both calls end up joined together. With abstract GC, this precision loss is avoided, as $\mathbb{Q}x$ is no longer reachable after returning from the first call to *id* (i.e., $\mathbb{Q}x \notin \widehat{\mathcal{R}}(\hat{\varsigma}_7^{\widehat{\Gamma}})$), and is therefore removed from the store by garbage collection (i.e., $\mathbb{Q}x \notin \widehat{\mathcal{S}}(\hat{\varsigma}_7^{\widehat{\Gamma}})$). As a result, when the second call to *id* binds its argument to $\mathbb{Q}x$, it does not need to join the new argument value with the old one,

and $@x$ is bound only to $\hat{v}_g$ (instead of $\hat{v}_g \sqcup \hat{v}_f$). Therefore, the binding of g in $\hat{\zeta}_{13}^{\hat{\Gamma}}$ is also precise, and the abstract interpreter knows exactly which closure is called. This avoids the need to branch into multiple successor states, eliminating the spurious path that occurred without abstract GC. As a result, $\hat{\mathcal{S}}(e_A) = \{\hat{\zeta}_i^{\hat{\Gamma}}\}_{i=0}^{16}$, and the final result of the abstract interpreter is $\widehat{\text{eval}}(e_A) = \hat{v}_{20} = (\emptyset, \text{cst}(20))$.

Note the importance of applying abstract GC between the first and second call to *id*: after binding $\hat{v}_g$, the address $@x$ once again becomes reachable, at which point it is too late to recover the precision loss by applying $\hat{\Gamma}$. That is, the interpreter can no longer know which of both closures stored at $@x$ is garbage.

2.3.3 Abstract Counting

One can further improve the precision of the abstract interpreter using *abstract counting* [106, 71, 13]. Specifically, the abstract interpreter currently loses precision for each variable assignment (rule E-SET) as it always has to apply a *weak update*: the new value of the variable is joined with its old value in the store. In general, any update to a mutable reference should perform a weak update. The main reason for applying such weak updates only is that an abstract address may represent multiple concrete addresses. As such, an update to *some* address represented by an abstract address must not overwrite the value for *all* addresses represented by that abstract address. A join is therefore necessary to maintain a sound over-approximation for the value stored at that abstract address.

However, if we knew that an abstract address only represents a single concrete address, then an update to that address can only correspond to an update to that concrete address. In this case, we could instead apply a *strong update*: the new value of the variable can safely overwrite the old value in the store. In general, when representing only a single concrete address, equality of abstract addresses implies equality for the corresponding concrete addresses. It is therefore safe for an update to such an address to effectively update the same address in the store, since they are guaranteed to represent the same address in the concrete.

Abstract counting extends the abstract interpreter to keep track of an *abstract count* for every address in the store. Such an abstract count provides a finite abstraction for the number of concrete addresses that are over-approximated by that address (or, equivalently, the number of times this address has been allocated). Since it only matters whether or not an abstract address represents a single concrete address, we use the following abstract counts: **0** (for an abstract address that has not yet been allocated), **1** (for an abstract address that has only

been allocated exactly once, i.e., representing a single concrete address) and ∞ (for an abstract address that has been allocated once or more, i.e., potentially representing multiple concrete addresses). As such, $\hat{\Sigma}$ is updated as follows:

$$\hat{\sigma} \in \widehat{\text{Store}} = \widehat{\text{Addr}} \rightarrow \widehat{\text{Val}} \times \widehat{\text{Count}} \quad \hat{q} \in \widehat{\text{Count}} ::= \mathbf{0} \mid \mathbf{1} \mid \infty$$

The abstraction $\widehat{\text{Count}}$ forms a lattice with the following definitions:

$$\perp_{\widehat{\text{Count}}} = \mathbf{0} \quad \mathbf{0} \sqsubseteq \mathbf{1} \sqsubseteq \infty \quad \hat{q}_1 \sqcup \hat{q}_2 = \begin{cases} \hat{q}_2 & \text{if } \hat{q}_1 \sqsubseteq \hat{q}_2 \\ \hat{q}_1 & \text{otherwise} \end{cases}$$

The abstract count of an address increases whenever the abstract address is used for a new allocation. For this purpose, we define a function $\widehat{\text{inc}} : \widehat{\text{Count}} \rightarrow \widehat{\text{Count}}$:

$$\widehat{\text{inc}}(\hat{q}) = \begin{cases} \mathbf{1} & \text{if } \hat{q} = \mathbf{0} \\ \infty & \text{otherwise} \end{cases}$$

Consequently, we modify the definitions of the store operations $\widehat{\text{lookup}}$, $\widehat{\text{extend}}$ and $\widehat{\text{update}}$ to take into account (and make use of) these abstract counts.

let $(\hat{v}, \hat{q}) = \hat{\sigma}(\hat{a})$:

$$\begin{aligned} \widehat{\text{lookup}}(\hat{\sigma}, \hat{a}) &= \hat{v} \\ \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}') &= \hat{\sigma}[\hat{a} \mapsto (\hat{v} \sqcup \hat{v}', \widehat{\text{inc}}(\hat{q}))] \\ \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v}') &= \begin{cases} \hat{\sigma}[\hat{a} \mapsto (\hat{v}', \mathbf{1})] & \text{if } \hat{q} = \mathbf{1} \\ \hat{\sigma}[\hat{a} \mapsto (\hat{v} \sqcup \hat{v}', \infty)] & \text{if } \hat{q} = \infty \end{cases} \end{aligned}$$

Extending the store (using $\widehat{\text{extend}}$) at an address increases the count of that address, as it corresponds to a new allocation (using that address). Updating the store (using $\widehat{\text{update}}$) at an existing address does not increase its abstract count. If the count at this address is $\mathbf{1}$, then we can safely apply a strong update, replacing the old value in the store with the new one (and therefore increasing the precision of the E-SET rule). However, if the count is ∞ , the address may represent

multiple concrete addresses, and therefore we need to fall back to a weak update. Note that the case where $\hat{q} = \mathbf{0}$ is not defined¹⁶ for $\widehat{\text{update}}$, as an update to an unallocated address is not allowed and should not happen by construction.

Impact on Precision. Abstract counting offers a direct precision benefit to the abstract interpreter because of strong updates. In addition, it also synergises well with abstract garbage collection. Without abstract GC, abstract counts can only increase. With abstract GC, addresses are collected once they become unreachable, and as a result their abstract counts are effectively reset to $\mathbf{0}$ (increasing the potential for future strong updates). In turn, strong updates remove the old value from the store, potentially reducing the set of reachable addresses $\widehat{\mathcal{R}}(\xi)$ (by removing addresses that were reachable only through that value) and therefore increasing the effectiveness of abstract garbage collection.

Impact on Performance. Similar to abstract garbage collection, abstract counting increases the precision of the abstract interpreter. These increases in precision can similarly improve performance as the exploration of spurious paths (which in general arise from any imprecision in control-flow behaviour) is avoided.

Abstract counting slightly increases the complexity of the analysis, as it directly increases the size of the state space $\widehat{\Sigma}$ to explore. A practical implementation may opt to disable abstract counting for addresses that are known to never be updated (e.g., addresses of immutable variables).

Note that abstract counting, similar to abstract GC, breaks the monotonicity guarantee for stores (i.e., stores no longer always form an ascending chain on each execution path). The reason for this is that a strong update may overwrite a value $\hat{v}_1$ with a new value $\hat{v}_2$ where $\hat{v}_1 \not\sqsubseteq \hat{v}_2$, so that the new store does not subsume the old one. Hence, in general the ACC no longer suffices to guarantee termination for the abstract interpreter; instead, the abstract domain needs to be finite.

Example. We illustrate the impact of abstract counting, in combination with abstract GC, on the program shown in Figure 2.8. Function f assigns 0 to its argument x and then returns x . The program returns 0 under concrete evaluation. As in the previous examples, we assume that the abstract interpreter uses $\mathbf{CP}_{\text{Num}}$ as the abstract numerical domain and is context-insensitive. The state exploration of the abstract interpreter using the $(\widehat{\rightarrow}_{\hat{\Gamma}})$ transition relation is shown below:

¹⁶This implies that $\widehat{\text{update}}$ is technically a partial function.

$\text{let } f = \lambda x.(\text{let } u = \text{set } x := 0 \text{ in } x) \text{ in}$	$e_E = \text{let } f = \lambda x.e_H \text{ in } e_F$
$\text{let } a = f(1) \text{ in}$	$e_F = \text{let } a = f(1) \text{ in } e_G$
$f(2)$	$e_G = f(2)$
	$e_H = \text{let } u = \text{set } x := 0 \text{ in } x$

 Figure 2.8: Program e_E (left) and the names of its subexpressions (right).

$$\begin{aligned}
 \widehat{\text{inject}}(e_E) &= \hat{\varsigma}_0 = \langle \mathbf{ev}(e_E, [], \epsilon), [], [], \hat{a}_{\text{halt}} \rangle \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_1 &= \langle \mathbf{ev}(\lambda x.e_H, [], \epsilon), [], \hat{\sigma}_k^1, \mathbb{Q}_k f \rangle & [\text{E-LET}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_2 &= \langle \mathbf{val}(\hat{v}_f), [], \hat{\sigma}_k^1, \mathbb{Q}_k f \rangle & [\text{E-ATOM}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_3 &= \langle \mathbf{ev}(e_F, \hat{\rho}_f, \epsilon), \hat{\sigma}^1, [], \hat{a}_{\text{halt}} \rangle & [\text{E-CONTINUE}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_4 &= \langle \mathbf{ev}(f(1), \hat{\rho}_f, \epsilon), \hat{\sigma}^1, \hat{\sigma}_k^2, \mathbb{Q}_k a \rangle & [\text{E-LET}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_5 &= \langle \mathbf{call}(\lambda x.e_H, [], \hat{v}_1, \epsilon), \hat{\sigma}^1, \hat{\sigma}_k^2, \mathbb{Q}_k a \rangle & [\text{E-CALL}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_6 &= \langle \mathbf{ev}(e_H, \hat{\rho}_x, \epsilon), \hat{\sigma}^2, \hat{\sigma}_k^2, \mathbb{Q}_k a \rangle & [\text{E-APP}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_7 &= \langle \mathbf{ev}(\text{set } x := 0, \hat{\rho}_x, \epsilon), \hat{\sigma}^2, \hat{\sigma}_k^3, \mathbb{Q}_k u \rangle & [\text{E-LET}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_8 &= \langle \mathbf{val}(\hat{v}_0), \hat{\sigma}^3, \hat{\sigma}_k^3, \mathbb{Q}_k u \rangle & [\text{E-SET}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_9 &= \langle \mathbf{ev}(x, \hat{\rho}_{x,u}, \epsilon), \hat{\sigma}^4, \hat{\sigma}_k^2, \mathbb{Q}_k a \rangle & [\text{E-CONTINUE}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{10} &= \langle \mathbf{val}(\hat{v}_0), \hat{\sigma}^1, \hat{\sigma}_k^2, \mathbb{Q}_k a \rangle & [\text{E-ATOM}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{11} &= \langle \mathbf{ev}(e_G, \hat{\rho}_{f,a}, \epsilon), \hat{\sigma}^5, [], \hat{a}_{\text{halt}} \rangle & [\text{E-CONTINUE}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{12} &= \langle \mathbf{call}(\lambda x.e_H, [], \hat{v}_2, \epsilon), [], [], \hat{a}_{\text{halt}} \rangle & [\text{E-CALL}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{13} &= \langle \mathbf{ev}(e_H, \hat{\rho}_x, \epsilon), \hat{\sigma}^6, [], \hat{a}_{\text{halt}} \rangle & [\text{E-APP}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{14} &= \langle \mathbf{ev}(\text{set } x := 0, \hat{\rho}_x, \epsilon), \hat{\sigma}^6, \hat{\sigma}_k^4, \mathbb{Q}_k u \rangle & [\text{E-LET}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{15} &= \langle \mathbf{val}(\hat{v}_0), \hat{\sigma}^7, \hat{\sigma}_k^4, \mathbb{Q}_k u \rangle & [\text{E-SET}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{16} &= \langle \mathbf{ev}(x, \hat{\rho}_{x,u}, \epsilon), \hat{\sigma}^8, [], \hat{a}_{\text{halt}} \rangle & [\text{E-CONTINUE}] \\
 \rhd_{\hat{\Gamma}} \hat{\varsigma}_{17} &= \langle \mathbf{val}(\hat{v}_0), [], [], \hat{a}_{\text{halt}} \rangle & [\text{E-ATOM}]
 \end{aligned}$$

$$\begin{aligned}
 \text{where } \hat{v}_f &= (\{(\lambda x.e_H, [])\}, \perp_{\text{CP}_{\text{Num}}}) & \hat{v}_n &= (\emptyset, \text{cst}(n)) \\
 \hat{\kappa}_0 &= \{(f, e_F, [], \epsilon, \hat{a}_{\text{halt}})\} & \hat{\sigma}_k^1 &= [\mathbb{Q}_k f \mapsto \hat{\kappa}_0] \\
 \hat{\kappa}_1 &= \{(a, e_G, \hat{\rho}_f, \epsilon, \hat{a}_{\text{halt}})\} & \hat{\sigma}_k^2 &= [\mathbb{Q}_k a \mapsto \hat{\kappa}_1] \\
 \hat{\kappa}_2 &= \{(u, x, \hat{\rho}_x, \epsilon, \mathbb{Q}_k a)\} & \hat{\sigma}_k^3 &= \hat{\sigma}_k^2[\mathbb{Q}_k u \mapsto \hat{\kappa}_2] \\
 \hat{\kappa}_3 &= \{(u, x, \hat{\rho}_x, \epsilon, \hat{a}_{\text{halt}})\} & \hat{\sigma}_k^4 &= [\mathbb{Q}_k u \mapsto \hat{\kappa}_3]
 \end{aligned}$$

$$\hat{\rho}_{x_1, \dots, x_n} = [x_1 \mapsto \mathbb{Q}x_1, \dots, x_n \mapsto \mathbb{Q}x_n]$$

$$\begin{aligned}
 \hat{\sigma}^1 &= [\mathbb{Q}f \mapsto (\hat{v}_f, 1)] & \hat{\sigma}^5 &= \hat{\sigma}^1[\mathbb{Q}a \mapsto (\hat{v}_0, 1)] \\
 \hat{\sigma}^2 &= \hat{\sigma}^1[\mathbb{Q}x \mapsto (\hat{v}_1, 1)] & \hat{\sigma}^6 &= [\mathbb{Q}x \mapsto (\hat{v}_2, 1)] \\
 \hat{\sigma}^3 &= \hat{\sigma}^2[\mathbb{Q}x \mapsto (\hat{v}_0, 1)] & \hat{\sigma}^7 &= \hat{\sigma}^6[\mathbb{Q}x \mapsto (\hat{v}_0, 1)] \\
 \hat{\sigma}^4 &= \hat{\sigma}^3[\mathbb{Q}u \mapsto (\hat{v}_0, 1)] & \hat{\sigma}^8 &= \hat{\sigma}^7[\mathbb{Q}u \mapsto (\hat{v}_0, 1)]
 \end{aligned}$$

Both calls to f use the same address $\mathbb{Q}x$ for their argument binding. In the first call to f , $\mathbb{Q}x$ is allocated for the first time, and therefore gets an abstract count of **1** in $\hat{\varsigma}_6$. Therefore, the variable assignment to x can perform a strong update in $\hat{\varsigma}_7$, replacing $\hat{v}_1$ with $\hat{v}_0$. Abstract GC then removes $\mathbb{Q}x$ from the store in $\hat{\varsigma}_{10}$, resetting its abstract count to **0**. As a result, the second call to f can again freshly allocate $\mathbb{Q}x$, so that its abstract count is again **1** (instead of ∞). The second call then also performs a strong update in $\hat{\varsigma}_{14}$, so that $\widehat{\text{eval}}(e_E) = \hat{v}_0 = (\emptyset, \text{cst}(0))$, which corresponds to the precise result of the program.

Note that both abstract GC and abstract counting are necessary to obtain this precise result. Without abstract counting, the assignments in both calls to f can only perform weak updates, immediately binding $\mathbb{Q}x$ to $\hat{v}_1 \sqcup \hat{v}_0 = \hat{v}_2 \sqcup \hat{v}_0 = (\emptyset, \top_{\mathbf{CP}_{\text{Num}}})$. Without abstract GC¹⁷, the second call to f would increase the abstract count of $\mathbb{Q}x$ to ∞ , so that the assignment in the second call can only perform a weak update, again resulting in $(\emptyset, \top_{\mathbf{CP}_{\text{Num}}})$.

2.3.4 Abstract Store Widening

The program semantics for λ_{ANF} expressed in terms of the transition relation ($\hat{\Rightarrow}$) (Section 2.3) are always computable. When using only finite abstract domains, one can show that the entire state space $\hat{\Sigma}$ is finite. The exploration of the abstract interpreter therefore always terminates, even when abstract GC and/or abstract counting are used. When using an infinite abstract domain (such as $\mathbf{CP}_{\text{Num}}$), the exploration only terminates when the stores grow monotonically on every evaluation path. This is notably not the case when using abstract GC and/or strong updates, since these break the assumption of “store monotonicity”.

However, even when using finite domains, the exploration of all states may be too computationally expensive. Most notably, each state $\hat{\varsigma}$ in $\hat{\Sigma}$ contains its own store, and the size of $\widehat{\text{Store}}$ alone is in the order of $\mathcal{O}(|\widehat{\text{Val}}|^{\text{Addr}})$, resulting in a potentially very large set of states that need to be explored.

Store widening offers a standard solution to these scalability issues and guarantees convergence regardless of the aforementioned assumption of store

¹⁷Furthermore, without abstract GC, both calls to f would also reuse the same continuation address $\mathbb{Q}_k u$, causing the abstract interpreter to also branch to a spurious return flow path from the second call to f to the first call site.

monotonicity [105]. Store widening joins stores of different states together. Since a store is a finite mapping from addresses to abstract values, the ACC for $\widehat{\text{Val}}$ implies the ACC for $\widehat{\text{Store}}$. Thus, by the ACC, an infinite sequence of stores $\hat{\sigma}_0, \hat{\sigma}_1, \dots$ from different states can be turned into a monotonically increasing sequence of stores $(\hat{\sigma}'_n)_{n \in \mathbb{N}}$ where $\hat{\sigma}'_0 = \hat{\sigma}_0$ and $\hat{\sigma}'_{i+1} = \hat{\sigma}'_i \sqcup \hat{\sigma}_{i+1}$, which is guaranteed to converge (to $\bigsqcup_{i \in \mathbb{N}} \hat{\sigma}'_i$) as $\hat{\sigma}'_i \sqsubseteq \hat{\sigma}'_{i+1}$ by design.

We integrate *global* store widening as formulated for AAM-based abstract interpreters by Might et al. [105]. Under global store widening, all states share a single global store (i.e., the stores of all states are effectively joined together).¹⁸

First, we introduce $\widehat{\text{reconstruct}} : \mathcal{P}(\widehat{\text{Control}} \times \widehat{\text{KAddr}}) \times \widehat{\text{Store}} \times \widehat{\text{KStore}} \rightarrow \mathcal{P}(\widehat{\Sigma})$, an auxiliary function which given a set of “storeless” states, reconstructs states in $\widehat{\Sigma}$ by injecting the given (global) store (and continuation store).

$$\widehat{\text{reconstruct}}((\hat{C}, \hat{\sigma}, \hat{\sigma}_k)) = \{ \langle \hat{c}, \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \mid \langle \hat{c}, \hat{a}_k \rangle \in \hat{C} \}$$

Next, we modify the computation of the program semantics, which is now defined using a least fixpoint iteration. This iteration keeps track of explored (“storeless”) states and the current global store(s). In each iteration, the global store is updated by joining in the stores of newly explored states.

$$\begin{aligned} \hat{S}(e) &= \widehat{\text{reconstruct}}(\text{lfp}(f)) \\ \textbf{where } \langle \hat{c}_0, \hat{\sigma}_0, \hat{\sigma}_{k_0}, \hat{a}_{k_0} \rangle &= \widehat{\text{inject}}(e) \\ f((\hat{C}, \hat{\sigma}, \hat{\sigma}_k)) &= (\hat{C}', \hat{\sigma}', \hat{\sigma}'_k) \\ \textbf{where } \hat{Q}' &= \{ \hat{c}' \mid \hat{c} \in \widehat{\text{reconstruct}}((\hat{C}, \hat{\sigma}, \hat{\sigma}_k)) \wedge \hat{c} \rhd \hat{c}' \} \\ \hat{C}' &= \{ \langle \hat{c}_0, \hat{a}_{k_0} \rangle \} \cup \{ \langle \hat{c}, \hat{a}_k \rangle \mid \langle \hat{c}, -, -, \hat{a}_k \rangle \in \hat{Q}' \} \\ \hat{\sigma}' &= \hat{\sigma}_0 \sqcup \hat{\sigma} \sqcup \bigsqcup \{ \hat{\sigma}' \mid \langle -, \hat{\sigma}', -, - \rangle \in \hat{Q}' \} \\ \hat{\sigma}'_k &= \hat{\sigma}_{k_0} \sqcup \hat{\sigma}_k \sqcup \bigsqcup \{ \hat{\sigma}'_k \mid \langle -, -, \hat{\sigma}'_k, - \rangle \in \hat{Q}' \} \end{aligned}$$

Impact on Precision. Store widening negatively impacts precision, as the stores of different states are now joined together (and joining lowers precision). In the case of global store widening, all values at the same address, regardless of which state they are bound in, are effectively joined together in a single store. This results in a *flow-insensitive* analysis, as the abstract interpreter does not discern

¹⁸We apply store widening to both the value store as well as the continuation store.

different stores at different program points. Thus, not only all stores on a single evaluation path, but also stores along different non-deterministic branches (which may exist for the sake of sound over-approximation) share the same store.

Impact on Performance. In general, store widening positively impacts performance and scalability (both in terms of running time and memory consumption). It has been shown that for a monovariant (context-insensitive) analysis, global store widening renders the complexity of the analysis cubic (as opposed to exponential) in terms of the program size [105]. In practice, it is often considered a must to realistically scale the analysis for larger programs.

Note that in rare cases, store widening may be detrimental to performance: just as increased precision may improve performance (as previously exemplified in Section 2.3.2), lower precision may lead to decreased performance when the imprecision causes the analysis to explore spurious paths.

Notably, global store widening is not compatible with either abstract GC or abstract counting: abstract GC is unable to collect any garbage (since the global store must preserve all bindings that are reachable from any state), whereas strong updates are never applicable (since all abstract counts eventually become ∞ when the global store converges).

3 Abstract Reference Counting

The benefits of abstract garbage collection include substantial improvements to both the precision and performance of the abstract interpreter. These benefits are maximised when the abstract interpreter adheres to the *garbage-free* property; i.e., when *all* garbage is collected *instantly and continuously*. Section 2.3.2 detailed the prototypical design of an abstract interpreter that is garbage-free by construction, as it applies abstract tracing GC at every evaluation step. However, realising this property with such a design causes its own significant performance overhead.

In this chapter, we develop a more efficient design for garbage-free abstract interpreters, based on reference counting instead of tracing GC (and therefore referred to as *abstract reference counting*). That is, instead of traversing the entire heap to compute $\hat{\mathcal{R}}(\xi)$ at every evaluation step (as with tracing GC), the main idea is to keep track of all references to every address $\hat{l}$ in $\hat{\mathcal{S}}(\xi)$ (as with reference counting), from which it is possible to determine whether $\hat{l} \in \hat{\mathcal{R}}(\xi)$. When there are no more references to $\hat{l}$, we have that $\hat{l} \notin \hat{\mathcal{R}}(\xi)$, and the memory location can be reclaimed. The converse, however, is not always true due to cyclic garbage. A direct application of reference counting (Section 3.2) therefore does not yet result in a garbage-free abstract interpreter. In fact, we observe that for an abstract interpreter, cyclic garbage occurs more frequently due to cyclic structures that are created because of the abstraction. We refer to these cycles as *artificial cycles*. In Section 3.3, we augment abstract reference counting with an algorithm to detect such cycles. Subsequently, we lift reference counting to the level of strongly connected components (instead of individual addresses).

Formally, we show that the resulting abstract interpreter is garbage-free and equivalent to one using $(\hat{\rightrightarrows}_{\hat{\Gamma}})$. We argue that such cycle detection adds little to the overall overhead for an abstract interpreter, and confirm this empirically in our experiments. In fact, we formally show that for the continuation store, where most of the cycles occur, the cycle detection technique only requires $\mathcal{O}(1)$ amortised operations per continuation push. Our evaluation (Section 3.4) shows that abstract reference counting (with cycle detection) achieves the optimal precision of abstract GC (i.e., it is garbage-free). It achieves significantly better performance compared to $(\hat{\rightrightarrows}_{\hat{\Gamma}})$, and both better precision and performance compared to other existing approaches that apply abstract GC less frequently.

3.1 Motivation

As discussed in Section 2.3.2, garbage can severely disrupt the behaviour of an abstract interpreter, negatively impacting both its precision and performance. Precision is lost when the abstract interpreter reallocates an unreachable address that is not yet collected, as the new value is joined with the old (garbage) value at that address (hence bringing the garbage “back to life”). This imprecision in data-flow can propagate rapidly and — since data flow impacts control flow (especially for higher-order languages [105]) — lead to imprecision in the control-flow behaviour of the abstract interpreter. That is, the imprecision may cause the abstract interpreter to explore *spurious program paths*, which can never occur in a concrete execution of the program. Performance is lost because the abstract interpreter unnecessarily analyses such spurious paths, and also because the interpreter explores equivalent states multiple times because they differ only due to garbage bindings in the store (i.e., for states $\hat{\varsigma}_1$ and $\hat{\varsigma}_2$ where $\hat{\varsigma}_1 \neq \hat{\varsigma}_2$ but $\hat{\Gamma}(\hat{\varsigma}_1) = \hat{\Gamma}(\hat{\varsigma}_2)$).

Abstract garbage collection can keep the interpreter’s memory, and hence the state space it explores, free of garbage and its negative effects. It has been shown to reduce the explored state space by orders of magnitude [107, 122], resulting in equally large improvements in precision and performance. These improvements are maximised when the abstract interpreter is entirely garbage-free (cf. Definition 1), as garbage can no longer have any impact on the abstract interpreter.

3.1.1 Problem: Application Policies for Abstract GC

As illustrated in the example of Section 2.3.2, garbage collection often needs to be triggered at a specific program point, as later GC applications can no longer recover the precision loss. Of course, the abstract interpreter is not privy to this knowledge when it arrives at that program point. Abstract GC therefore ought to be applied preemptively; the question is only *how often* and *when*. Garbage will build up and impact precision and performance when GC is applied infrequently, while too frequent applications bring about their own performance overhead. A balance between these tradeoffs is often difficult to strike. Therefore, an abstract interpreter may use one of several policies to decide *when to apply abstract GC*.

Survey of Existing Policies. A simple, but surprisingly effective policy that is often used in practice [107, 118, 143, 40, 45] is to apply abstract GC at every evaluation step of the abstract interpreter (as with $(\rightrightarrows_{\hat{\Gamma}})$). This policy results in a garbage-free abstract interpreter, therefore achieving the maximum precision and performance benefits of abstract GC. However, realising this property by applying abstract GC at every step comes at a cost, as the GC overhead itself can become

performance-detrimental, in particular for programs with large heaps [88, 45]. Since the size of the heap is generally proportional to the size of the program, the cost of abstract GC becomes prohibitive for larger programs.

Other policies apply GC less frequently to avoid this overhead, which does result in some precision loss from non-collected garbage (i.e., the abstract interpreter is no longer completely garbage-free). For instance, the TAJIS static analysis [73] (for JavaScript) applies abstract GC upon every function exit. This policy requires fewer GC applications, and function exits are good GC application candidates as garbage tends to be left behind when exiting a function’s scope. TAJIS in fact uses a more lightweight (but also more conservative) approach to abstract GC¹: instead of traversing the entire heap after every function return, GC only collects garbage bindings that were allocated during the current invocation of the function that was returned to. Objects that existed before this invocation are kept alive, even when they have become garbage. This also makes abstract GC cheaper to apply, but further prevents garbage-freeness (regardless of the application policy). The MOPSA analysis framework has been used for the analysis of Python programs [111] where abstract GC is applied after each assignment for which the right-hand side is a method call. The authors report that they have also experimented with application policies that invoke abstract GC for only a fraction of these assignments (to lower its performance overhead), but that this rendered the impact of abstract GC unsatisfactory. The policy of Γ CFA [106] (and later, LFA [104]), which pioneered the use of abstract GC, is to apply abstract GC when the address allocator returns a memory location that is already in use. As such, it triggers abstract GC whenever values need to be joined in the store, so that by design a value can never be joined with garbage. While the Γ CFA policy avoids a common form of garbage-induced imprecision, it still applies GC frequently. The explored state space can moreover not be considered garbage-free, as some of the explored states remain polluted with garbage. This results in redundant evaluations of equivalent states that differ only in garbage bindings. In fact, in a follow-up paper [107], the authors reverted their policy to an application of GC at every evaluation step for the sake of garbage-freeness. In general, any policy that does not apply garbage collection at every step is no longer entirely garbage-free, at best leading to the exploration of garbage-polluted states, and at worst severely deteriorating the analysis’ precision and performance.

Precision Impact of Policies. Figure 3.1 illustrates the tradeoff between the frequency of GC applications and precision for 3 different GC application policies: no GC (as in $(\Rightarrow)$), GC at every join operation in the store (as in Γ CFA) and GC

¹Motivated by its use of lazy propagation (see Section 3.5 for more details).

at every step (as in $(\hat{\Rightarrow}_{\hat{\Gamma}})$). We compare the size of the state space explored by an abstract interpreter for a sample Scheme program (specifically, the `collatz` benchmark used in Section 3.4). A smaller state space corresponds to a higher precision of the analysis, as fewer (spurious) states have to be explored.

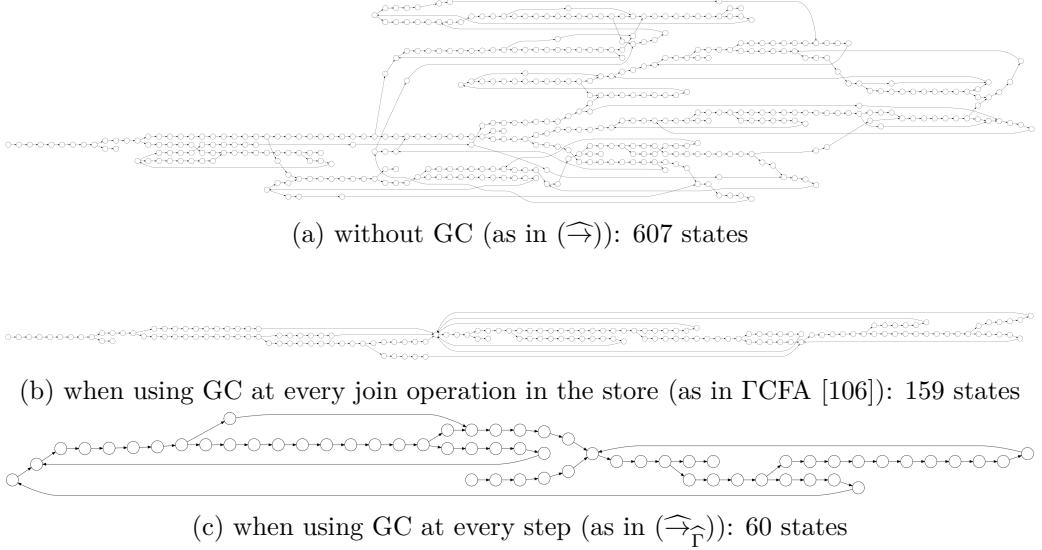


Figure 3.1: States explored by the abstract interpreter for different GC policies.

Policies that apply GC more frequently reduce the state space to be explored, which results in both higher precision and performance, but at the same time also causes a performance overhead. Our evaluation in Section 3.4 shows that this overhead significantly decreases the throughput of the abstract interpreter. Policies that apply GC less frequently (e.g., the Γ CFA one) often prove more efficient in terms of performance despite the larger state space to explore. In terms of precision, applying GC at every step yields the optimal result.

3.1.2 Approach: Abstract Reference Counting

We aim to realise garbage-free abstract interpretation more efficiently, attaining optimal precision without significant compromises to performance. We propose to collect garbage in an abstract interpreter using a form of reference counting. For a concrete interpreter, tracing GC (such as the mark-and-sweep and stop-and-copy algorithms) is usually deemed superior to reference counting, as the latter may result in a greater overhead (in terms of both memory usage and performance) [144] and cannot deal with cycles. For an abstract interpreter, we argue that reference counting needs to be re-evaluated with respect to tracing GC.

Reconsidering the Disadvantages of Reference Counting. The overhead of reference counting is less pronounced for an abstract interpreter. The reason is two-fold. First, abstract interpreters typically use fewer addresses than their concrete counterparts, as a single abstract address in general corresponds to multiple concrete addresses. Thus, fewer reference counts need to be maintained. Second, abstract interpreters can tolerate more memory management overhead than concrete interpreters, as their performance is usually dominated by other factors.

The fact that reference counting cannot reclaim cyclic garbage, however, remains an issue in the abstract. In fact, we have observed cyclic garbage to be more prevalent in the abstract. Section 3.3 addresses this problem separately as it directly impedes realising the garbage-free property using reference counting.

Reconsidering the Advantages of Reference Counting. The main advantage of reference counting stems from its continuous nature. Garbage is collected under reference counting as soon as it arises. As soon as the reference count of an address becomes zero, reference counting immediately reclaims that address. Tracing GC approaches hold on to garbage until the next GC application. Unless the application requires minimal GC pauses, this is generally not an issue for concrete interpreters. For abstract interpreters, however, garbage held on to impacts precision — requiring abstract tracing GCs to be applied frequently. Simple bookkeeping suffices for abstract reference counting to immediately and efficiently eliminate all garbage, precluding the need for GC application policies that seek to balance the imprecision of long-lived garbage with the overhead of collection frequency. It collects garbage continuously before it can cause a precision loss.

Finally, tracing GC needs to traverse the entire heap, which can get larger for more complex programs, resulting in poor analysis scalability. Such expensive traversals increase overhead when applied frequently. In contrast, the cost of reference counting is mostly insensitive to the size of the heap [11]; the cost of collecting garbage is only proportional to the amount of garbage collected.

3.2 Reference Counting for Abstract GC

In this section, we develop an alternative, more efficient design for garbage-free abstract interpreters. As the abstract interpreter performs abstract garbage collection through reference counting, we refer to it as *abstract reference counting*. We augment the abstract semantics for λ_{ANF} (cf. Section 2.3) with reference counting, and show that, in the absence of cycles, it is equivalent to a garbage-free abstract interpreter using abstract tracing GC with the transition relation $(\rightrightarrows_{\hat{\Gamma}})$. Section 3.3 discusses our solution to the problem of cyclic garbage.

3.2.1 Abstract Interpretation with Reference Counting using $(\Rightarrow_{\text{arc}})$

In order to incorporate reference counting into the abstract interpreter, it is necessary to know the *reference count* for each abstract address $\hat{l}$; that is, the number of abstract addresses that reference $\hat{l}$ (according to the adjacency relation $(\rightsquigarrow_{\xi})$). For an abstract interpreter, we do not directly maintain the reference count $(\mathbb{N})^2$ for each address $\hat{l}$, but rather the set of addresses $(\mathcal{P}(\widehat{\text{Loc}}))$ that reference $\hat{l}$ directly. The reference count of $\hat{l}$ can then be derived as the cardinality of this set.

State Space. We augment the abstract state space to keep track of this set of addresses for every address by adding a component $\hat{\phi}_{\xi}$ to every state $\hat{\varsigma}$. We label this state space $\widehat{\Sigma}_{\text{arc}}$ to differentiate it from the original state space $\widehat{\Sigma}$. Figure 3.2 shows the definition of the state space $\widehat{\Sigma}_{\text{arc}}$; we elide definitions for components that preserve the same definition as in $\widehat{\Sigma}$ (cf. Figure 2.5).

$$\hat{\varsigma} \in \widehat{\Sigma}_{\text{arc}} ::= \langle \hat{c}, \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \qquad \hat{\phi} \in \widehat{\text{Refs}} = \widehat{\text{Loc}} \rightarrow \mathcal{P}(\widehat{\text{Loc}})$$

Figure 3.2: The updated state space $\widehat{\Sigma}_{\text{arc}}$ of the abstract interpreter with reference counting for λ_{ANF} .

Tracking the set of addresses $\hat{\phi}(\hat{l})$ instead of the actual reference count $|\hat{\phi}(\hat{l})|$ renders our formalisation more concise (e.g., by avoiding the double counting of references) and amenable to the incorporation of cycle detection (Section 3.3).

Note that this extension of the state space explored by the abstract interpreter does not actually increase its complexity, as the added component $\hat{\phi}_{\xi}$ can be defined deterministically in terms of $\hat{\varsigma}$'s original components. That is, we modify the transition relation to maintain the following invariant for every state $\hat{\varsigma}$:

$$\hat{\phi}_{\xi}(\hat{l}) = \{\hat{l}' \in \widehat{\text{Loc}} \mid \hat{l}' \rightsquigarrow_{\xi} \hat{l}\}$$

In other words, we maintain $\hat{\phi}$ so that $\hat{\phi}(\hat{l})$ contains all addresses that refer directly to $\hat{l}$ (or rather, all addresses whose value/continuation directly refers to $\hat{l}$). To this end, we adapt the store-modifying functions $\widehat{\text{extend}}$ and $\widehat{\text{update}}$ to update $\hat{\phi}$ accordingly for each modification in the store. Both functions now take the current $\hat{\phi}$ as an extra argument, and return an updated $\hat{\phi}$ as part of their result. In

²Note that it is not necessary to maintain a finite approximation such as $\widehat{\mathbb{N}} = \{0, 1, \infty\}$ for this reference count. The key insight is to count the “abstract” references to an abstract address, not the concrete references to the corresponding concrete address(es) that are approximated by that abstract address. That is, our approach performs abstract interpretation *with* reference counting, not an abstract interpretation *of* reference counting.

addition, $\widehat{\text{update}}$ also returns a set of addresses $\hat{D} \subseteq \widehat{\text{Loc}}$ for which the reference count *may* have decreased. This can happen in the case of a strong update, as references originating from the “old” overwritten value are removed from the store.

We introduce the operator $\ominus : \widehat{\text{Refs}} \rightarrow \widehat{\text{Refs}} \rightarrow \widehat{\text{Refs}}$ as a convenient notation for removing references from $\hat{\phi}$, defined as $(\hat{\phi}_1 \ominus \hat{\phi}_2)(\hat{l}) = \hat{\phi}_1(\hat{l}) \setminus \hat{\phi}_2(\hat{l})$. The updated definitions of $\widehat{\text{extend}}$ and $\widehat{\text{update}}$ are given below.³

$$\begin{aligned}
 & \text{let } (\hat{v}, \hat{q}) = \hat{\sigma}(\hat{a}) : \\
 & \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}', \hat{\phi}) = (\hat{\sigma}[\hat{a} \mapsto (\hat{v} \sqcup \hat{v}', \widehat{\text{inc}}(\hat{q}))], \hat{\phi} \sqcup \bigsqcup_{\hat{l} \in \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}')} [\hat{l} \mapsto \{\hat{a}\}]) \\
 & \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v}', \hat{\phi}) \\
 & = \begin{cases} (\hat{\sigma}[\hat{a} \mapsto (\hat{v}', \mathbf{1})], \hat{\phi}' \sqcup \bigsqcup_{\hat{l} \in \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}')} [\hat{l} \mapsto \{\hat{a}\}], \hat{D}) & \text{if } \hat{q} = \mathbf{1} \\ (\hat{\sigma}[\hat{a} \mapsto (\hat{v} \sqcup \hat{v}', \infty)], \hat{\phi} \sqcup \bigsqcup_{\hat{l} \in \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}')} [\hat{l} \mapsto \{\hat{a}\}], \emptyset) & \text{if } \hat{q} = \infty \end{cases} \\
 & \text{where } \hat{D} = \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}) \\
 & \hat{\phi}' = \hat{\phi} \ominus \bigsqcup_{\hat{l} \in \hat{D}} [\hat{l} \mapsto \{\hat{a}\}]
 \end{aligned}$$

When extending the store at address $\hat{a}$ with value $\hat{v}'$, all addresses reachable from $\hat{v}'$ become reachable from $\hat{a}$; hence, for all addresses directly reachable from $\hat{v}'$ we add $\hat{a}$ as a reference in $\hat{\phi}$.⁴ When updating the store at address $\hat{a}$ with value $\hat{v}'$, there are two possibilities. In case of a weak update ($\hat{q} = \infty$), we proceed as with $\widehat{\text{extend}}$, and return with $\hat{D} = \emptyset$ as no references were removed to any address. In case of a strong update ($\hat{q} = \mathbf{1}$), references from the overwritten value $\hat{v}$ may no longer be reachable from $\hat{a}$; hence, we first remove $\hat{a}$ from all addresses directly reachable from $\hat{v}$, and then proceed by adding $\hat{a}$ to all addresses reachable from the new value $\hat{v}'$. We return with $\hat{D} = \widehat{\mathcal{T}}_{\text{Val}}(\hat{v})$, as these addresses may have lost a reference from $\hat{a}$ (note that in general $\widehat{\mathcal{T}}_{\text{Val}}(\hat{v})$ and $\widehat{\mathcal{T}}_{\text{Val}}(\hat{v}')$ may overlap).

Similarly, we modify function $\widehat{\text{extend}}_k$ to adapt $\hat{\phi}$, registering references for every address that is reachable from the newly allocated continuation.

³We slightly abuse notation and write $[a \mapsto b]$ for singleton maps (i.e., $[a \mapsto b] = \perp[a \mapsto b]$).

⁴Note that this suffices since $\widehat{\mathcal{T}}_{\text{Val}}(\hat{v} \sqcup \hat{v}') = \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}) \cup \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}')$.

$$\widehat{\text{extend}}_k(\hat{\sigma}_k, \hat{a}_k, \hat{\kappa}, \hat{\phi}) = (\hat{\sigma}_k[\hat{a}_k \mapsto \hat{\sigma}_k(\hat{a}_k) \cup \hat{\kappa}], \hat{\phi} \sqcup \bigsqcup_{\hat{l} \in \widehat{\mathcal{T}}_{\text{Kont}}(\hat{\kappa})} [\hat{l} \mapsto \{\hat{a}_k\}])$$

Transition Relation. We introduce the new transition relation ($\widehat{\Rightarrow}_{\text{arc}}$) as a composition of an auxiliary transition relation ($\widehat{\Rightarrow}_0$) and a function $\widehat{\text{collect}}$.

$$\frac{\hat{\varsigma} \widehat{\Rightarrow}_0 (\hat{\varsigma}', \hat{D}) \quad \hat{\varsigma}'' = \widehat{\text{collect}}(\hat{\varsigma}', \widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\varsigma}) \cup \hat{D})}{\hat{\varsigma} \widehat{\Rightarrow}_{\text{arc}} \hat{\varsigma}''}$$

The auxiliary relation ($\widehat{\Rightarrow}_0$) extends the transition relation ($\widehat{\Rightarrow}$) of Figure 2.6 by updating the references of addresses (in $\hat{\phi}$) for every update to the store (or continuation store). Analogous to $\widehat{\text{update}}$, it determines the set of addresses $\hat{D} \subseteq \widehat{\text{Loc}}$ for which the reference count *may* have decreased after the evaluation step (i.e., we have that $\forall \hat{\varsigma}, \hat{\varsigma}', \hat{l} : \hat{\varsigma} \widehat{\Rightarrow}_0 (\hat{\varsigma}', \hat{D}) \wedge \hat{\phi}_{\hat{\varsigma}}(\hat{l}) \not\subseteq \hat{\phi}_{\hat{\varsigma}'}(\hat{l}) \implies \hat{l} \in \hat{D}$).

The function $\widehat{\text{collect}}$ is responsible for collecting garbage as reference counts become zero (i.e., $|\hat{\phi}_{\hat{\varsigma}'}(\hat{l})| = 0$ for some $\hat{l} \notin \widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\varsigma}')$) after every step of the transition relation ($\widehat{\Rightarrow}_0$). It checks all addresses that may have their reference counts decreased: this includes the “root set” of addresses $\widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\varsigma})$ that were previously directly reachable from $\hat{\varsigma}$ (and may no longer be reachable or have any references in $\hat{\phi}_{\hat{\varsigma}'}$), as well as the set of addresses $\hat{D}$ returned by ($\widehat{\Rightarrow}_0$).

Figure 3.3 shows the definition of the auxiliary transition relation ($\widehat{\Rightarrow}_0$). The most important modification — compared to transition relation ($\widehat{\Rightarrow}$) of Figure 2.6 — is that each transition now passes along a (possibly updated) $\hat{\phi}$ to correctly maintain reference counts between addresses. Rules that do not modify the store (i.e., rules E-ATOM, E-CALL and E-BINOP) make no modifications to $\hat{\phi}$ (and accordingly return with $\hat{D} = \emptyset$). Rules that modify the store using $\widehat{\text{update}}$ or $\widehat{\text{extend}}$ (i.e., rules E-SET, E-APP and E-CONTINUE) return the updated $\hat{\phi}'$ (and $\hat{D}$) that are in turn returned by these functions. Since rule E-LET updates the continuation store, we update $\hat{\phi}$ (analogously to $\widehat{\text{extend}}$) to update the reference counts of all addresses directly reachable from the new continuation.

$$\begin{array}{c}
 \frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle \mathbf{ev}(ae, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \mathbf{val}(\hat{v}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle, \emptyset)} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad (\hat{\sigma}', \hat{\phi}', \hat{D}) = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v}, \hat{\phi})}{\langle \mathbf{ev}(\text{set } x := ae, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \mathbf{val}(\hat{v}), \hat{\sigma}', \hat{\sigma}_k, \hat{a}_k, \hat{\phi}' \rangle, \hat{D})} \quad [\text{E-SET}] \\
 \\
 \frac{\begin{array}{c} (\hat{f}, _) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \\ (\text{lam}, \hat{\rho}') \in \hat{f} \quad \hat{t}' = \text{tick}(f, \hat{t}) \quad \hat{c} = \mathbf{call}(\text{lam}, \hat{\rho}', \hat{v}, \hat{t}') \end{array}}{\langle \mathbf{ev}(f(ae), \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \hat{c}, \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle, \emptyset)} \quad [\text{E-CALL}] \\
 \\
 \frac{\hat{a} = \widehat{\text{alloc}}(x, \hat{t}) \quad \hat{\rho}' = \hat{\rho}[x \mapsto \hat{a}] \quad (\hat{\sigma}', \hat{\phi}') = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}, \hat{\phi})}{\langle \mathbf{call}(\lambda x.e, \hat{\rho}, \hat{v}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \mathbf{ev}(e, \hat{\rho}', \hat{t}), \hat{\sigma}', \hat{\sigma}_k, \hat{a}_k, \hat{\phi}' \rangle, \emptyset)} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{c} (_, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (_, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\ \hat{v} = (\emptyset, \hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \quad \hat{c} = \mathbf{val}(\hat{v}) \end{array}}{\langle \mathbf{ev}(op(ae_1, ae_2), \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \hat{c}, \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle, \emptyset)} \quad [\text{E-BINOP}] \\
 \\
 \frac{\begin{array}{c} \hat{a}'_k = \widehat{\text{alloc}}_k(e_1, \hat{t}) \quad \hat{\kappa} = \{(x, e_2, \hat{\rho}, \hat{t}, \hat{a}_k)\} \\ \hat{c} = \mathbf{ev}(e_1, \hat{\rho}, \hat{t}) \quad (\hat{\sigma}'_k, \hat{\phi}') = \widehat{\text{extend}}_k(\hat{\sigma}_k, \hat{a}_k, \hat{\kappa}, \hat{\phi}) \end{array}}{\langle \mathbf{ev}(\text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \hat{c}, \hat{\sigma}, \hat{\sigma}'_k, \hat{a}'_k, \hat{\phi}' \rangle, \emptyset)} \quad [\text{E-LET}] \\
 \\
 \frac{\begin{array}{c} (x, e, \hat{\rho}, \hat{t}, \hat{a}'_k) \in \hat{\sigma}_k(\hat{a}_k) \quad \hat{a} = \widehat{\text{alloc}}(x, \hat{t}) \\ \hat{\rho}' = \hat{\rho}[x \mapsto \hat{a}] \quad (\hat{\sigma}', \hat{\phi}') = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}, \hat{\phi}) \end{array}}{\langle \mathbf{val}(\hat{v}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k, \hat{\phi} \rangle \rhd_0 (\langle \mathbf{ev}(e, \hat{\rho}', \hat{t}), \hat{\sigma}', \hat{\sigma}_k, \hat{a}'_k, \hat{\phi}' \rangle, \emptyset)} \quad [\text{E-CONTINUE}]
 \end{array}$$

 Figure 3.3: Transition relation ($\rhd_0$) for abstract reference counting in λ_{ANF} .

While ($\rhd_0$) maintains $\hat{\phi}$ as intended, it does not yet collect any garbage. Recall that garbage is defined as all addresses that are not reachable, i.e., all addresses that are not in $\widehat{\mathcal{R}}(\hat{\varsigma}) = \{\hat{l}' \in \widehat{\text{Loc}} \mid \exists \hat{l} \in \widehat{\text{Loc}} : \hat{l} \in \widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\varsigma}) \wedge \hat{l} \xrightarrow{*}_{\hat{\varsigma}} \hat{l}'\}$. We refer to $\widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\varsigma})$ as the *root set* of addresses for a given state $\hat{\varsigma}$. Whenever $\hat{\varsigma}$ transitions to $\hat{\varsigma}'$,

the root set changes from $\widehat{\mathcal{T}}_{\Sigma}(\xi)$ to $\widehat{\mathcal{T}}_{\Sigma}(\xi')$, which creates garbage whenever some address $\hat{l}$ that was in $\widehat{\mathcal{T}}_{\Sigma}(\xi)$ is no longer in $\widehat{\mathcal{T}}_{\Sigma}(\xi')$ and when $\hat{l}$ is not referenced from any other address (i.e., when $|\hat{\phi}_{\xi'}(\hat{l})| = 0$).⁵

Next to changes in the root set, strong updates can also create garbage (see above), since they remove a value from the store and its outgoing references to other addresses from $\hat{\phi}$. Addresses that were previously directly referenced by this overwritten value (i.e., the set $\hat{D}$ returned by $\widehat{\text{update}}$ and then propagated by the auxiliary transition relation ($\widehat{\rightarrow}_0$)) may lose a reference, and therefore may need to be garbage collected if their reference count has become zero.

Function $\widehat{\text{collect}}$, defined below⁶, therefore needs to collect all garbage that is created either due to a change in the root set $\widehat{\mathcal{T}}_{\Sigma}(\xi)$ or that originates from the set $\hat{D}$ of addresses potentially orphaned because of a strong update.

$$\begin{aligned} \widehat{\text{collect}}(\xi, \hat{D}) &= \begin{cases} \xi & \text{if } \hat{D} = \emptyset \\ \widehat{\text{check}}(\hat{l}, \xi, \hat{D} \setminus \{\hat{l}\}) & \exists \hat{l} \in \hat{D} \end{cases} \\ \widehat{\text{check}}(\hat{l}, \xi, \hat{D}) &= \begin{cases} \widehat{\text{dealloc}}(\hat{l}, \xi, \hat{D}) & \text{if } |\hat{\phi}_{\xi}(\hat{l})| = 0 \wedge \hat{l} \notin \widehat{\mathcal{T}}_{\Sigma}(\xi) \\ \widehat{\text{collect}}(\xi, \hat{D}) & \text{otherwise} \end{cases} \\ \widehat{\text{dealloc}}(\hat{l}, \xi, \hat{D}) &= \widehat{\text{collect}}(\xi', \hat{D} \cup \hat{T}) \\ \text{where } \hat{T} &= \{\hat{l}' \mid \hat{l} \widehat{\rightsquigarrow}_{\xi} \hat{l}'\} \\ \xi' &= \langle \hat{c}_{\xi}, \hat{\sigma}_{\xi} \setminus \{\hat{l}\}, \hat{\sigma}_{k_{\xi}} \setminus \{\hat{l}\}, \hat{a}_{k_{\xi}}, \hat{\phi} \ominus \bigsqcup_{\hat{l}' \in \hat{T}} [\hat{l}' \mapsto \{\hat{l}\}] \rangle \end{aligned}$$

For every address $\hat{l}$ in $\hat{D}$, function $\widehat{\text{collect}}$ checks whether it is still referenced by at least one other address (i.e., if $|\hat{\phi}_{\xi}(\hat{l})| > 0$) or whether it is still part of the new root set (i.e., $\hat{l} \in \widehat{\mathcal{T}}_{\Sigma}(\xi)$). If this is not the case, then it should garbage collect $\hat{l}$ by removing it from the store. Garbage collecting an address $\hat{l}$ also removes $\hat{l}$ from $\hat{\phi}(\hat{l}')$ for every $\hat{l}' \in \widehat{\text{Loc}}$ where $\hat{l} \widehat{\rightsquigarrow}_{\xi} \hat{l}'$. This is akin to decrementing the reference count of all addresses $\hat{l}'$ referenced by $\hat{l}$. The update can in turn cause other addresses to be garbage collected because they no longer have any references. Therefore, $\widehat{\text{collect}}$ adds these addresses to $\hat{D}$ to recursively check them as well.

One can verify from the definitions of ($\widehat{\rightarrow}_0$) and $\widehat{\text{collect}}$ that the $\hat{\phi}$ -component is

⁵Note that $\hat{\phi}$ only takes into account references coming from other addresses in the store (and continuation store); root references therefore have to be handled separately.

⁶We use the notation for set removal to remove elements from a map m , so that $(m \setminus S)(x) = \perp$ if $x \in S$ and $(m \setminus S)(x) = m(x)$ if $x \notin S$.

always updated correctly. That is, for every state $\hat{\zeta} \in \hat{\Sigma}$, the reference-counting transition relation $(\rightrightarrows_{\text{arc}})$ maintains the invariant $\hat{\phi}_{\hat{\zeta}}(\hat{l}) = \{\hat{l}' \mid \hat{l}' \rightsquigarrow_{\hat{\zeta}} \hat{l}\}$, so that we can reason about $\hat{\phi}(\hat{l})$ as the set of addresses that refer to $\hat{l}$.

Program Semantics. We update the injection function to start the abstract interpreter with an empty $\hat{\phi}$ component⁷, as the stores are also initially empty.

$$\widehat{\text{inject}}(e) = \langle \mathbf{ev}(e, [], \hat{t}_0), \perp_{\widehat{\text{Store}}}, \perp_{\widehat{\text{KStore}}}, \hat{a}_{\text{halt}}, \perp_{\widehat{\text{Refs}}} \rangle$$

Subsequently, we can define the abstract collecting semantics for λ_{ANF} with abstract reference counting. We define $\hat{S}_{\text{arc}} : \text{Exp} \rightarrow \mathcal{P}(\hat{\Sigma}_{\text{arc}})$ as follows:

$$\hat{S}_{\text{arc}}(e) = \{\hat{\zeta} \in \hat{\Sigma}_{\text{arc}} \mid \widehat{\text{inject}}(e) \xrightarrow{*}_{\text{arc}} \hat{\zeta}\}$$

3.2.2 Properties of $(\rightrightarrows_{\text{arc}})$

An abstract interpreter that applies abstract tracing GC (as in Section 2.3.2), but not at every step, is GC sound, but no longer garbage-free (cf. Definitions 1 and 2). We now show that — in the absence of cycles — using $(\rightrightarrows_{\text{arc}})$ as a transition relation results in a GC sound and garbage-free abstract interpreter without the need to trigger a full tracing GC at every evaluation step.

First, Theorem 4 shows that abstract reference counting is GC sound.

Theorem 4 $(\rightrightarrows_{\text{arc}})$ is GC sound.

Proof. Assuming $\hat{\zeta}$ is GC sound and $\hat{\zeta} \xrightarrow{\text{arc}} \hat{\zeta}'$, by inversion of the definition of $(\rightrightarrows_{\text{arc}})$ we have that $\exists \hat{\zeta}_0, \hat{D}$ so that $\hat{\zeta} \xrightarrow{0} (\hat{\zeta}_0, \hat{D})$ and $\hat{\zeta}' = \widehat{\text{collect}}(\hat{\zeta}_0, \widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\zeta}) \cup \hat{D})$. To prove the GC soundness of $(\rightrightarrows_{\text{arc}})$, we have to show that $\hat{\zeta}'$ is also GC sound.

A trivial case analysis shows that $(\xrightarrow{0})$ is GC sound (since everything it adds to $\widehat{\mathcal{R}}(\hat{\zeta})$ is also added to $\widehat{\mathcal{S}}(\hat{\zeta})$). Therefore, by definition $\hat{\zeta}_0$ is also GC sound.

We show that whenever $\widehat{\text{collect}}$ collects an address $\hat{l}$, we have that $\hat{l} \notin \widehat{\mathcal{R}}(\hat{\zeta}_0)$. By the definition of $\widehat{\text{check}}$, $\hat{l}$ can only be collected when $\hat{l} \notin \widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\zeta}_0)$ and when every address $\hat{l}' \in \hat{\phi}_{\hat{\zeta}_0}(\hat{l})$ has already been collected. By induction, we can derive that every such address $\hat{l}' \notin \widehat{\mathcal{R}}(\hat{\zeta}_0)$. Therefore, since $\hat{l}$ is not directly reachable from the root set $\widehat{\mathcal{T}}_{\hat{\Sigma}}(\hat{\zeta}_0)$ or any other address in $\widehat{\mathcal{R}}(\hat{\zeta}_0)$, we have that $\hat{l} \notin \widehat{\mathcal{R}}(\hat{\zeta}_0)$.

⁷Note that $\widehat{\text{Refs}}$ forms a lattice with $\perp_{\widehat{\text{Refs}}} = \lambda \hat{l}. \emptyset$.

Since $\widehat{\mathcal{S}}_0$ is GC sound, we have that $\widehat{\mathcal{R}}(\widehat{\mathcal{S}}_0) \subseteq \widehat{\mathcal{S}}(\widehat{\mathcal{S}}_0)$. As we have just shown that $\widehat{\text{collect}}$ only removes addresses from $\widehat{\mathcal{S}}_0$ that are not in $\widehat{\mathcal{R}}(\widehat{\mathcal{S}}_0)$, we get that $\widehat{\mathcal{R}}(\widehat{\mathcal{S}}_0) \subseteq \widehat{\mathcal{S}}(\zeta')$ and $\widehat{\mathcal{R}}(\zeta') = \widehat{\mathcal{R}}(\widehat{\mathcal{S}}_0)$, so that $\widehat{\mathcal{R}}(\zeta') \subseteq \widehat{\mathcal{S}}(\zeta')$. Hence, ζ' is GC sound. $\square$

Our current formulation of abstract reference counting is GC sound, but in general not yet garbage-free, as cyclic garbage cannot be collected. Theorem 5 therefore states that $(\rightrightarrows_{\text{arc}})$ is garbage-free only in the absence of cycles.

For convenience, we define $\hat{\phi}_{\zeta}^*(\hat{l}) = \{\hat{l}' \mid \hat{l}' \rightsquigarrow_{\zeta}^* \hat{l}\} \setminus \{\hat{l}\}$, so that $\hat{\phi}_{\zeta}^*(\hat{l})$ is the set of all addresses that can directly or indirectly reach $\hat{l}$ (excluding $\hat{l}$ itself).

Theorem 5 *In the absence of cycles, $(\rightrightarrows_{\text{arc}})$ is garbage-free.*

Proof. Assuming $\hat{\zeta}$ is garbage-free and $\hat{\zeta} \rightrightarrows_{\text{arc}} \zeta'$, by inversion of the definition of $(\rightrightarrows_{\text{arc}})$ we have that $\exists \hat{\mathcal{S}}_0, \hat{D}$ so that $\hat{\zeta} \rightrightarrows_0 (\widehat{\mathcal{S}}_0, \hat{D})$ and $\zeta' = \widehat{\text{collect}}(\widehat{\mathcal{S}}_0, \widehat{\mathcal{T}}_{\widehat{\mathcal{S}}}(\hat{\zeta}) \cup \hat{D})$. To prove that $(\rightrightarrows_{\text{arc}})$ is garbage-free, we have to show that ζ' is also garbage-free.

Consider $\hat{l} \in \widehat{\mathcal{S}}(\zeta')$. We need to prove that $\hat{l} \in \widehat{\mathcal{R}}(\zeta')$. There are two possibilities:

- $\hat{l} \notin \widehat{\mathcal{S}}(\hat{\zeta})$: in this case, $\hat{l}$ was added to the store (or continuation store) by one of the transition rules in $(\rightrightarrows_0)$. Using a trivial case analysis, it is clear that in any of those cases, we also have that $\hat{l} \in \widehat{\mathcal{R}}(\zeta')$.
- $\hat{l} \in \widehat{\mathcal{S}}(\hat{\zeta})$: in this case, by the garbage-freeness of $\hat{\zeta}$, we have that $\hat{l} \in \widehat{\mathcal{R}}(\hat{\zeta})$.

Proceed by induction on $|\hat{\phi}_{\zeta'}^*(\hat{l})|$.

- For the case $|\hat{\phi}_{\zeta'}^*(\hat{l})| = 0$, by definition of $\hat{\phi}^*$ and $\hat{\phi}$ we also have that $|\hat{\phi}_{\zeta'}(\hat{l})| = 0$. To show that $\hat{l} \in \widehat{\mathcal{T}}_{\widehat{\mathcal{S}}}(\zeta')$ (and hence, $\hat{l} \in \widehat{\mathcal{R}}(\zeta')$), it suffices to show that $\hat{l}$ was “checked” (yet not removed, hence $\hat{l} \in \widehat{\mathcal{T}}_{\widehat{\mathcal{S}}}(\zeta')$) using the function $\widehat{\text{check}}$. If we already had that $|\hat{\phi}_{\zeta}(\hat{l})| = 0$, then $\hat{l}$ was checked by $\widehat{\text{check}}$ because it must have been that $\hat{l} \in \widehat{\mathcal{T}}_{\widehat{\mathcal{S}}}(\hat{\zeta})$ (since $\hat{l} \in \widehat{\mathcal{R}}(\hat{\zeta})$); if $|\hat{\phi}_{\zeta}(\hat{l})| > 0$, then $\hat{l}$ was also checked, either because references to $\hat{l}$ were removed during $\widehat{\text{collect}}$, or because references were removed due to a strong update, both of which leading to $\hat{l}$ being added to $\hat{D}$.
- For the case $|\hat{\phi}_{\zeta'}^*(\hat{l})| = k > 0$, we have that $|\hat{\phi}_{\zeta'}(\hat{l})| > 0$. In the absence of cycles, references between addresses form a directed acyclic graph (DAG). Pick any predecessor $\hat{l}' \in \hat{\phi}_{\zeta'}(\hat{l})$ in the DAG; A predecessor $\hat{l}'$ of an address $\hat{l}$ in the DAG always has fewer ancestors than $\hat{l}$, i.e., $|\hat{\phi}_{\zeta'}(\hat{l}')| < k$, so that — by the induction hypothesis — $\hat{l}' \in \widehat{\mathcal{R}}(\zeta')$. Since $\hat{l}' \in \widehat{\mathcal{R}}(\zeta')$ and $\hat{l}' \rightsquigarrow_{\zeta'} \hat{l}$ (by definition of $\hat{\phi}_{\zeta'}^*(\hat{l})$), we have that $\hat{l} \in \widehat{\mathcal{R}}(\zeta')$. Hence, $\widehat{\mathcal{S}}(\zeta') \subseteq \widehat{\mathcal{R}}(\zeta')$, i.e., ζ' is garbage-free.

$\square$

Note that, as abstract reference counting is both GC sound and garbage-free (in the absence of cycles), we have that $\widehat{\mathcal{R}}(\hat{\xi}) = \widehat{\mathcal{S}}(\hat{\xi})$.

Normally, we would still have to prove other properties of $(\rightrightarrows_{\text{arc}})$ (such as the soundness of the abstract interpretation); instead, we prove an equivalence with the existing transition relation $(\rightrightarrows_{\widehat{\Gamma}})$, for which such properties have already been established in related work [168, 106]. First, we introduce a function $\widehat{rc} : \widehat{\Sigma} \rightarrow \widehat{\Sigma}_{\text{arc}}$ to map states from the abstract interpreter using the transition relation $(\rightrightarrows_{\widehat{\Gamma}})$ to equivalent states in the updated state space $\widehat{\Sigma}_{\text{arc}}$:

$$\widehat{rc}(\hat{\xi}) = \langle \hat{c}_{\hat{\xi}}, \hat{\sigma}_{\hat{\xi}}, \hat{\sigma}_{k\hat{\xi}}, \hat{a}_{k\hat{\xi}}, \hat{\phi} \rangle \quad \text{where } \hat{\phi}(\hat{l}) = \{\hat{l}' \in \widehat{\text{Loc}} \mid \hat{l}' \rightsquigarrow_{\hat{\xi}} \hat{l}\}$$

Theorem 6 *In the absence of cycles, the transition relation $(\rightrightarrows_{\text{arc}})$ is equivalent to $(\rightrightarrows_{\widehat{\Gamma}})$ in the sense that: $\forall \hat{\xi}, \hat{\xi}' \in \widehat{\Sigma}, \hat{\xi} \rightrightarrows_{\widehat{\Gamma}} \hat{\xi}' \iff \widehat{rc}(\hat{\xi}) \rightrightarrows_{\text{arc}} \widehat{rc}(\hat{\xi}')$.*

Proof. It is clear that the transition rules of $(\rightrightarrows_0)$ are isomorphic to those of $(\rightrightarrows)$, and that equivalence holds between both transition relations in that $\forall \hat{\xi}, \hat{\xi}_0 \in \widehat{\Sigma}, \hat{\xi} \rightrightarrows \hat{\xi}_0 \iff \exists \hat{D} : \widehat{rc}(\hat{\xi}) \rightrightarrows_0 (\widehat{rc}(\hat{\xi}_0), \hat{D})$. Both $(\rightrightarrows_{\text{arc}})$ and $(\rightrightarrows_{\widehat{\Gamma}})$ then only remove addresses from $\hat{\sigma}$ and $\hat{\sigma}_k$ (using $\widehat{\text{collect}}$ and $\widehat{\Gamma}$, respectively), and since we have already proven both GC sound and garbage-free, they produce equivalent states (since $\widehat{\mathcal{R}}(\hat{\xi}_0) = \widehat{\mathcal{R}}(\widehat{rc}(\hat{\xi}_0))$, while $\widehat{\mathcal{R}}(\hat{\xi}) = \widehat{\mathcal{S}}(\hat{\xi})$ in both cases). $\square$

Corollary 3 *In the absence of cycles, $\forall e. \hat{S}_{\text{arc}}(e) = \{\widehat{rc}(\hat{\xi}) \mid \hat{\xi} \in \hat{S}(e)\}$.*

Proof. Follows directly from Theorem 6. $\square$

3.3 Reclaiming Garbage Cycles

We have shown that the $(\rightrightarrows_{\text{arc}})$ transition relation is garbage-free under the premise that no states are encountered with cycles in their store or continuation store. The underlying reason for this limitation is that pure reference counting cannot reclaim cycles [144]. However, it is clear that this premise does not hold in general. A concrete interpreter can end up with cycles in the store σ when a program creates cyclic data structures. In this case, abstract interpretation of this program will also result in a cyclic structure in $\hat{\sigma}$, which will never be reclaimed using $(\rightrightarrows_{\text{arc}})$. Hence, in general $(\rightrightarrows_{\text{arc}})$ is only GC sound, not garbage-free.

In this section, we argue that cycles are an even larger problem for an abstract interpreter, as additional *artificial cycles* can form due to the over-approximation (i.e., cycles that do not form in the concrete interpreter). We explain why (Section 3.3.1) and integrate a cycle detection algorithm into abstract reference counting (Section 3.3.2) to reclaim cyclic garbage. In Section 3.3.3, we show that this addition results in a garbage-free abstract interpreter, while only mildly increasing overhead (which we show formally for the continuation store).

3.3.1 Artificial Cycles in $(\rightrightarrows_{\text{arc}})$

We show that cycles are much more common for an abstract interpreter. That is, for a given program e , an abstract interpreter may encounter states with cycles in either $\hat{\sigma}$ (or $\hat{\sigma}_k$) that would not occur in σ (or σ_k) in the corresponding states explored by a concrete interpreter. The underlying reason is that an abstract interpreter can only use a finite number of addresses, and therefore is sometimes forced to reuse an address for different allocations. As the same abstract address is typically reused for different allocations at the same program point, recursively created structures often “collapse” and become cyclic after abstraction.

From a theoretical perspective, this implies that an abstract address ($\in \widehat{\text{Loc}}$) is used as the abstraction for multiple concrete addresses ($\in \text{Loc}$). We assume that abstraction function $\alpha : \text{Loc} \rightarrow \widehat{\text{Loc}}$ maps a concrete address l to the abstract address $\hat{l}$, while concretisation function $\gamma : \widehat{\text{Loc}} \rightarrow \mathcal{P}(\text{Loc})$ returns all concrete addresses that are abstracted to an abstract address. It can be shown from the abstract semantics [168] that if l_1 references l_2 (i.e., $l_1 \rightsquigarrow l_2$) under concrete interpretation, then $\alpha(l_1)$ references $\alpha(l_2)$ under abstract interpretation too (i.e., $\alpha(l_1) \rightsquigarrow \alpha(l_2)$). Now consider the linear sequence of concrete references $l_0 \rightsquigarrow l_1 \rightsquigarrow \dots \rightsquigarrow l_k$ where $l_i \neq l_j$ when $i \neq j$. Its abstraction $\alpha(l_0) \rightsquigarrow \alpha(l_1) \rightsquigarrow \dots \rightsquigarrow \alpha(l_k)$ becomes cyclic as soon as for some $i \neq j$, $\alpha(l_i) = \alpha(l_j)$. We refer to the cycles that lack a concrete counterpart and stem from the interpreter’s abstraction as *artificial cycles*. This situation is depicted in Figure 3.4.

In practice, it turns out that artificial cycles are mostly a problem in the continuation store. Consider the E-LET transition rule. This transition rule “pushes” a continuation on the continuation store, so that if $\varsigma \rightarrow \varsigma'$, we have that $a_{k\varsigma'} \rightsquigarrow a_{k\varsigma}$. As alloc_k always returns a fresh address under concrete interpretation, a sequence of transitions of this rule will give rise to $a_{kn} \rightsquigarrow a_{kn-1} \rightsquigarrow \dots \rightsquigarrow a_{k0}$ where $a_{kn} \neq a_{kn-1} \neq \dots \neq a_{k0}$. Under abstract interpretation, in contrast, the same sequence will result in $\alpha(a_{kn}) \rightsquigarrow \alpha(a_{kn-1}) \rightsquigarrow \dots \rightsquigarrow \alpha(a_{k0})$. A cycle will be created as soon as $\widehat{\text{alloc}}_k$ allocates the same address more than once in such a sequence (i.e., once $\alpha(a_{ki}) = \alpha(a_{kj})$ for some $0 \leq i < j \leq n$). This is usually the case when the abstract interpreter visits the same program location multiple times due to,

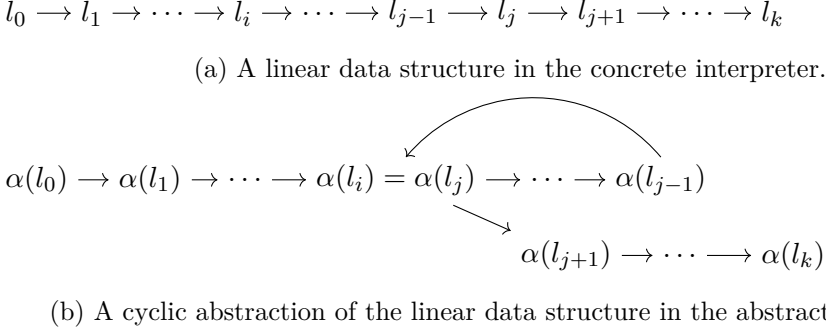


Figure 3.4: Example of artificial cycles arising due to abstraction.

for instance, looping behaviour. When the resulting cycle becomes unreachable, none of its constituent addresses will be reclaimed under $(\hat{\rhd}_{\text{arc}})$ from the continuation store where they cause imprecisions. While imprecision in the value store $\hat{\sigma}$ may be tolerable, imprecision in the continuation store $\hat{\sigma}_k$ has a direct detrimental impact on both precision and performance. The E-CONTINUE transition rule, for instance, will generate *spurious* successor states if the set $\hat{\sigma}_k(\hat{a}_k)$ contains “garbage continuations” due to such imprecision resulting in the exploration of infeasible control flow. This directly results in imprecise program paths (which can in turn indirectly further propagate imprecision), and harms performance by generating unnecessary exploration work for the abstract interpreter.

3.3.2 Abstract Reference Counting with Cycle Detection: $(\hat{\rhd}_{\text{arc}++})$

We extend $(\hat{\rhd}_{\text{arc}})$ with a cycle detection algorithm, so that it remains garbage-free in the presence of cycles. Existing techniques for reference counting to reclaim garbage cycles do not immediately help in this setting (Section 3.5). We therefore propose a domain-specific solution that is tailored towards the artificial cycles that are artefacts of abstraction. First, we make a few observations:

- As previously discussed, cycles are the most prevalent in the continuation store, where not reclaiming such cycles directly impacts both precision and performance negatively. Our main priority is therefore the continuation store. While realising the garbage-free property requires cycle detection in both stores, we show that its application to the continuation store does not have a significant performance overhead (Theorem 7).
- An artificial cycle can by definition only be created when an address is reused, and conversely, address reuse in practice often results in a cycle. In fact, in a garbage-free abstract interpreter, we show that reusing an address in the continuation store is guaranteed to create a cycle.

- Except for the [E-SET] rule, the transition rules in $(\widehat{\Rightarrow}_0)$ only make the stores grow monotonically. This implies that (in the absence of strong updates), existing cycles can only be merged together into a larger cycle.
- Elements (and therefore potentially cycles) are only removed from the store under two circumstances: when removed by a strong update in the [E-SET] rule, and when garbage collected by $\widehat{\text{collect}}$. In the former case, an existing cycle may “break up” into multiple smaller cycles (because outgoing references from the overwritten value are removed). In the latter case, the entire cycle is collected (because if some address that is part of a cycle is collected, then by definition of garbage the entire cycle should be collected).

Hence, while it is not obvious how to detect when a cycle becomes a garbage cycle, it is possible to predict the creation of a cycle efficiently. Therefore, our approach explicitly tracks all cycles — or rather *strongly connected components* (SCCs) — in $\widehat{\mathcal{S}}(\xi)$. Such cycles grow when SCCs are merged together, can be split upon a strong update, or be removed in their entirety when collected as garbage, enabling maintaining a disjoint-set of the SCCs in $\widehat{\mathcal{S}}(\xi)$ efficiently. The idea is then to maintain the reference count for every SCC (instead of for every individual address), only counting “external” references coming from other SCCs. When an SCC loses all of its external references, all addresses that are part of that SCC can be garbage collected. As a cycle can by definition never occur between two SCCs, applying reference counting at this level overcomes its main limitation.

State Space. Figure 3.5 depicts the updates to $\widehat{\Sigma}_{\text{arc}}$ to incorporate cycle detection. We elide definitions for components that preserve the same definition as the corresponding components in $\widehat{\Sigma}_{\text{arc}}$ (cf. Figure 3.2).

$$\begin{aligned} \widehat{\phi} \in \widehat{\text{Refs}} &= (\widehat{\text{SCC}} \multimap \mathcal{P}(\widehat{\text{Loc}})) \times \widehat{\text{DS}} \\ \widehat{\pi} \in \widehat{\text{DS}} &= \mathcal{P}(\widehat{\text{SCC}}) & \widehat{s}, \widehat{scc} \in \widehat{\text{SCC}} &= \mathcal{P}(\widehat{\text{Loc}}) \end{aligned}$$

Figure 3.5: Updated state space $\widehat{\Sigma}_{\text{arc}}$ of the abstract interpreter using reference counting with cycle detection for λ_{ANF} .

Newly added component $\widehat{\pi}$ tracks the partitioning of $\widehat{\mathcal{S}}(\xi)$ into its SCCs. The abstract interpreter needs to maintain disjoint-set $\widehat{\pi}$ so that $\bigcup_{\widehat{scc} \in \widehat{\pi}} \widehat{scc} = \widehat{\text{Loc}}$ and for any $\widehat{s}_1, \widehat{s}_2 \in \widehat{\pi}$ where $\widehat{s}_1 \neq \widehat{s}_2$, $\widehat{s}_1 \cap \widehat{s}_2 = \emptyset$ (i.e., $\widehat{\pi}$ is a partition of $\widehat{\text{Loc}}$). In what follows, we assume that functions `union` and `find` (for which efficient

implementations have been proposed [50]) exist, so that:

$$\text{union}(\hat{\pi}, \hat{s}_1, \hat{s}_2) = (\hat{\pi} \setminus \{\hat{s}_1, \hat{s}_2\}) \cup \{\hat{s}_1 \cup \hat{s}_2\}$$

$$\text{find}(\hat{\pi}, \hat{l}) = \widehat{scc} \text{ where } \widehat{scc} \in \hat{\pi} \wedge \hat{l} \in \widehat{scc}$$

The original definition of $\hat{\phi}$ from Section 3.2.1 also changes in that it now tracks all incoming references *for every strongly connected component* $\widehat{scc}$ from addresses that are not part of $\widehat{scc}$ (i.e., addresses that belong to some other SCC).

We extend the state-indexed shorthand notation so that $\hat{\phi}_\xi = (\hat{R}_\xi, \hat{\pi}_\xi)$. We also slightly abuse notation by writing $\hat{\phi}(\hat{l})$ for $\hat{R}(\text{find}(\hat{\pi}, \hat{l}))$ when $\hat{\phi} = (\hat{R}, \hat{\pi})$.

Note again that the state space does not increase in complexity, since $\hat{\pi}_\xi$, and therefore also $\hat{\phi}_\xi$, are defined deterministically in terms of ξ 's other components.

$$\hat{R}_\xi(\widehat{scc}) = \{\hat{l}' \in \widehat{\text{Loc}} \mid \hat{l}' \notin \widehat{scc} \wedge \exists \hat{l} \in \widehat{scc}, \hat{l}' \rightsquigarrow_\xi \hat{l}\} \text{ for } \widehat{scc} \in \hat{\pi}_\xi$$

$$\hat{\pi}_\xi = \{\{\hat{l}' \in \widehat{\text{Loc}} \mid \hat{l} \rightsquigarrow_\xi^* \hat{l}' \wedge \hat{l}' \rightsquigarrow_\xi^* \hat{l}\} \mid \hat{l} \in \widehat{\text{Loc}}\}$$

Due to the reflexivity of $(\rightsquigarrow_\xi^*)$, we define the bottom element $\perp_{\widehat{\text{Refs}}}$ as follows:

$$\perp_{\widehat{\text{Refs}}} = (\lambda \widehat{scc}. \emptyset, \{\{\hat{l}\} \mid \hat{l} \in \widehat{\text{Loc}}\})$$

Transition Relation. Transition relation $(\rightrightarrows_0)$ needs to be updated to maintain the $\hat{\pi}$ and $\hat{\phi}$ components. We write $(\rightrightarrows_{0++})$ for the updated transition relation. Likewise, we write $(\rightrightarrows_{\text{arc}++})$ for the updated transition relation (instead of $(\rightrightarrows_{\text{arc}})$) that makes use of the auxiliary transition relation $(\rightrightarrows_{0++})$ (instead of $(\rightrightarrows_0)$).

The evaluation rules for $(\rightrightarrows_{0++})$ are the same as those given in Figure 3.3; however, we adapt functions **extend** and **update** to ensure that (1) cycles are correctly tracked in $\hat{\pi}$ and (2) incoming references are counted per SCC (instead of per address). The updated definitions of **extend** and **update** are given below⁸:

let $(\hat{v}, \hat{q}) = \hat{\sigma}(\hat{a}) :$

$\widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}', \hat{\phi}) = (\hat{\sigma}[\hat{a} \mapsto (\hat{v} \sqcup \hat{v}', \widehat{\text{inc}}(\hat{q}))], \text{ADDEDGES}(\hat{a}, \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}'), \hat{\phi}))$

$\widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v}', \hat{\sigma}_k, \hat{\phi})$

⁸Note that function **update** now also receives the continuation store as an extra argument. This is necessary to correctly update the bookkeeping of references in $\hat{\phi}$ (cf. Algorithm 2). We omit the corresponding (trivial) adaptation in the E-SET rule of $(\rightrightarrows_{0++})$ for brevity.

$$\begin{aligned}
 &= \begin{cases} (\hat{\sigma}', \text{REPLACEEDGES}(\hat{a}, \hat{D}, \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}'), \hat{\sigma}', \hat{\sigma}_k, \hat{\phi}), \hat{D}) & \text{if } \hat{q} = 1 \\ (\hat{\sigma}[\hat{a} \mapsto (\hat{v} \sqcup \hat{v}', \infty)], \text{ADDEDGES}(\hat{a}, \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}'), \hat{\phi}), \emptyset) & \text{if } \hat{q} = \infty \end{cases} \\
 &\text{where } \hat{\sigma}' = \hat{\sigma}[\hat{a} \mapsto (\hat{v}', 1)] \\
 &\quad \hat{D} = \widehat{\mathcal{T}}_{\text{Val}}(\hat{v})
 \end{aligned}$$

Analogously, we adapt the definition of $\widehat{\text{extend}}_k$ to detect cycles in the continuation store using the same auxiliary function `ADDEDGES`.

$$\widehat{\text{extend}}_k(\hat{\sigma}_k, \hat{a}_k, \hat{\kappa}, \hat{\phi}) = (\hat{\sigma}_k[\hat{a}_k \mapsto \hat{\sigma}_k(\hat{a}_k) \cup \hat{\kappa}], \text{ADDEDGES}(\hat{a}_k, \widehat{\mathcal{T}}_{\text{Kont}}(\hat{\kappa}), \hat{\phi}))$$

Previously, whenever a reference from address $\hat{l}_{from}$ to a set of addresses $\hat{S}$ was added to the store, $\hat{\phi}$ was updated to $\hat{\phi} \sqcup \bigsqcup_{\hat{l}_{to} \in \hat{S}} [\hat{l}_{to} \mapsto \{\hat{l}_{from}\}]$. We now replace this update with the function call $\text{ADDEDGES}(\hat{l}_{from}, \hat{S}, \hat{\phi})$ (Algorithm 1) to check if SCCs are merged together when the store (or continuation store) is extended. For every new reference from $\hat{l}_{from}$ to $\hat{l}_{to}$ that is added to the store, it calls the function `ADDEDGE` to detect if a new cycle has been created and to update $\hat{\phi}$ and $\hat{\pi}$ accordingly (line 28). The `ADDEDGE` function initiates a backward search (using the helper function `SEARCH`) for the SCC of $\hat{l}_{to}$, starting from the SCC of $\hat{l}_{from}$ (line 21). All strongly connected components that can be traversed in a backwards path from $\hat{l}_{from}$ to $\hat{l}_{to}$ need to be merged together in $\hat{\pi}$ using `union` (line 18), since such a path implies that a cycle is created as $\hat{l}_{to} \xrightarrow{*} \hat{l}_{from}$ and $\hat{l}_{from} \rightsquigarrow \hat{l}_{to}$ (due to the new reference added to the store). During this traversal, `SEARCH` also keeps track of all incoming references to this newly created SCC (line 19), so that it can immediately update $\hat{R}$ as well (line 22). If no such path is found, $\hat{l}_{from}$ and $\hat{l}_{to}$ are not part of the same SCC, so that $\hat{\pi}$ remains unchanged and $\hat{\phi}$ is updated as previously (line 24). Note that the check $\widehat{scc} \in V$ (line 8) is only there to prevent redundant work when an SCC has already been visited through a different path; its purpose is *not* to prevent an infinite loop (for `SEARCH`), which cannot happen as there are no cyclic paths between SCCs.

Function `REPLACEEDGES` in Algorithm 2 is responsible for updating $\hat{\phi}$ when a strong update occurs in the store (note that strong updates only apply to the value store $\hat{\sigma}$). It takes as arguments the address $\hat{l}_{from}$ that is strongly updated, the set of addresses $\hat{D}$ previously referenced from that address by the overwritten value, and the set of addresses $\hat{S}$ of the new value at this address. First, it directly updates the bookkeeping in $\hat{R}$ (line 5) by removing all references from $\hat{l}_{from}$ to previously referenced addresses (i.e., addresses that are in $\hat{D}$ but no longer

ALGORITHM 1: Cycle detection in $(\widehat{\rightarrow}_{0++})$ when adding edges.

```

1  function ADDEGE( $\hat{l}_{from}, \hat{l}_{to}, \hat{\phi}$ ):
2       $V \leftarrow \emptyset$ 
3       $I \leftarrow \hat{\phi}(\hat{l}_{to})$ 
4       $\langle \hat{R}, \hat{\pi} \rangle \leftarrow \hat{\phi}$ 
5      function SEARCH( $\hat{l}$ ):
6           $\widehat{scc} \leftarrow \text{find}(\hat{\pi}, \hat{l})$ 
7          if  $\widehat{scc} = \text{find}(\hat{\pi}, \hat{l}_{to})$  then return true
8          if  $\widehat{scc} \in V$  then return false
9           $V \leftarrow V \cup \{\widehat{scc}\}$ 
10          $reachable \leftarrow \text{false}$ 
11          $incoming \leftarrow \emptyset$ 
12         foreach  $\hat{l}' \in \hat{R}(\widehat{scc})$  do
13             if SEARCH( $\hat{l}'$ ) then
14                  $reachable \leftarrow \text{true}$ 
15             else
16                  $incoming \leftarrow incoming \cup \{\hat{l}'\}$ 
17         if  $reachable$  then
18              $\hat{\pi} \leftarrow \text{union}(\hat{\pi}, \widehat{scc}, \text{find}(\hat{\pi}, \hat{l}_{to}))$ 
19              $I \leftarrow I \cup incoming$ 
20         return  $reachable$ 
21     if SEARCH( $\hat{l}_{from}$ ) then
22         return  $\langle \hat{R}|_{\hat{\pi}} \sqcup [\text{find}(\hat{\pi}, \hat{l}_{to}) \mapsto I], \hat{\pi} \rangle$ 
23     else
24         return  $\langle \hat{R} \sqcup [\text{find}(\hat{\pi}, \hat{l}_{to}) \mapsto \{\hat{l}_{from}\}], \hat{\pi} \rangle$ 
25 function ADDEDGES( $\hat{l}_{from}, \hat{S}, \hat{\phi}$ ):
26     foreach  $\hat{l}_{to} \in \hat{S}$  do
27         if  $\hat{l}_{from} \notin \hat{\phi}(\hat{l}_{to})$  then
28              $\hat{\phi} \leftarrow \text{ADDEGE}(\hat{l}_{from}, \hat{l}_{to}, \hat{\phi})$ 
29     return  $\hat{\phi}$ 

```

in $\hat{S}$). Next, it checks if a strongly connected component is no longer strongly connected after a strong update (hence breaking up into multiple SCCs). This can only occur if a reference was removed *within* a single SCC; the algorithm therefore checks whether the SCC of the strongly updated address $\hat{l}_{from}$ is itself one of the SCCs to which a reference was removed. In this case, we update the partitioning $\hat{\pi}$ by recomputing the strongly connected components for the addresses that were part of the original SCC (line 8). This can for instance be done using either Tarjan's [155] or Kosaraju's [2] algorithm. Note that the work required for this step is only proportional to the size of the SCC in which the strong update occurred, as these algorithms are only needed for the subset of the

ALGORITHM 2: Cycle detection in $(\Rightarrow_{0++})$ for strong updates.

```

1  function REPLACEEDGES( $\hat{l}_{from}, \hat{D}, \hat{S}, \hat{\sigma}, \hat{\sigma}_k, \hat{\phi}$ ):
2       $\langle \hat{R}, \hat{\pi} \rangle \leftarrow \hat{\phi}$ 
3       $\widehat{scc} \leftarrow \text{find}(\hat{\pi}, \hat{l}_{from})$ 
4       $removed \leftarrow \{\text{find}(\hat{\pi}, \hat{l}) \mid \hat{l} \in \hat{D} \setminus \hat{S}\}$ 
5       $\hat{R} \leftarrow \hat{R} \ominus \bigsqcup_{\widehat{scc}' \in removed} [\widehat{scc}' \mapsto \{\hat{l}_{from}\}]$ 
6      if  $\widehat{scc} \in removed$  then
7           $\hat{\sigma}' \leftarrow \hat{\sigma}|_{\widehat{scc}}$ 
8           $\hat{\pi} \leftarrow (\hat{\pi} \setminus \{\widehat{scc}\}) \cup \{\{\hat{l}' \in \widehat{scc} \mid \hat{l} \xrightarrow{*}_{\hat{\sigma}}, \hat{l}' \wedge \hat{l}' \xrightarrow{*}_{\hat{\sigma}'}, \hat{l}\} \mid \hat{l} \in \widehat{scc}\}$ 
9          foreach  $\hat{l} \in \hat{R}(\widehat{scc})$  do
10              $refs \leftarrow \begin{cases} \widehat{\mathcal{T}}_{\text{Val}}(\hat{\sigma}(\hat{l})) & \text{if } \hat{l} \in \widehat{\text{Addr}} \\ \widehat{\mathcal{T}}_{\text{Kont}}(\hat{\sigma}_k(\hat{l})) & \text{if } \hat{l} \in \widehat{\text{KAddr}} \end{cases}$ 
11              $\hat{R} \leftarrow \hat{R} \sqcup \bigsqcup_{\hat{l}' \in refs \cap \widehat{scc}} [\text{find}(\hat{\pi}, \hat{l}') \mapsto \{\hat{l}\}]$ 
12             foreach  $\hat{l}' \in \widehat{scc}$  do
13                  $refs \leftarrow \widehat{\mathcal{T}}_{\text{Val}}(\hat{\sigma}(\hat{l}'))$ 
14                  $\widehat{scc}' \leftarrow \text{find}(\hat{\pi}, \hat{l}')$ 
15                  $\hat{R} \leftarrow \hat{R} \sqcup \bigsqcup_{\hat{l} \in refs \setminus \widehat{scc}'} [\text{find}(\hat{\pi}, \hat{l}) \mapsto \{\hat{l}'\}]$ 
16              $\hat{R} \leftarrow \hat{R}|_{\hat{\pi}}$ 
17  return ADDEDGES( $\hat{l}_{from}, \hat{S}, \langle \hat{R}, \hat{\pi} \rangle$ )
    
```

store restricted to this SCC. After recomputing the SCCs, the algorithm updates the references recorded in $\hat{R}$ accordingly. First, it redistributes all references to the original SCC over the newly recomputed SCCs (lines 9–11). It does this by re-registering all *external* references, previously referencing the original SCC (i.e., $\hat{R}(\widehat{scc})$) to the updated SCC they now each reference. Second, it considers the references that were previously *internal* to the original SCC (and hence not registered in $\hat{R}$) and may now form (external) references between newly formed SCCs (lines 12–15). It does this by re-registering the outgoing references from each address in the original SCC to their updated target SCC (ensuring that internal references remain unregistered in $\hat{R}$). Finally, the algorithm takes into account potential newly formed SCCs due to new references in $\hat{S}$ by calling function ADDEDGES of Algorithm 1 (line 17), as is done for weak updates.

The only remaining modification is to adapt function $\widehat{\text{collect}}$ to work at the level of SCCs (instead of individual addresses).

$$\begin{aligned}
 \widehat{\text{collect}}(\hat{\varsigma}, \hat{D}) &= \widehat{\text{collect}}'(\hat{\varsigma}, \{\text{find}(\hat{\pi}_{\hat{\varsigma}}, \hat{l}) \mid \hat{l} \in \hat{D}\}) \\
 \widehat{\text{collect}}'(\hat{\varsigma}, \hat{D}) &= \begin{cases} \hat{\varsigma} & \text{if } \hat{D} = \emptyset \\ \widehat{\text{check}}(\widehat{scc}, \hat{\varsigma}, \hat{D} \setminus \{\widehat{scc}\}) & \exists \widehat{scc} \in \hat{D} \end{cases} \\
 \widehat{\text{check}}(\widehat{scc}, \hat{\varsigma}, \hat{D}) &= \begin{cases} \widehat{\text{dealloc}}(\widehat{scc}, \hat{\varsigma}, \hat{D}) & \text{if } |\hat{R}_{\hat{\varsigma}}(\widehat{scc})| = 0 \wedge \widehat{scc} \notin \widehat{RS} \\ \widehat{\text{collect}}'(\hat{\varsigma}, \hat{D}) & \text{otherwise} \end{cases} \\
 \text{where } \widehat{RS} &= \{\text{find}(\hat{\pi}_{\hat{\varsigma}}, \hat{l}) \mid \hat{l} \in \widehat{\mathcal{T}}_{\widehat{\Sigma}}(\hat{\varsigma})\} \\
 \widehat{\text{dealloc}}(\widehat{scc}, \hat{\varsigma}, \hat{D}) &= \widehat{\text{collect}}'(\hat{\varsigma}', \hat{D} \cup \hat{T}) \\
 \text{where } \hat{T} &= \{\text{find}(\hat{\pi}_{\hat{\varsigma}}, \hat{l}') \mid \exists \hat{l} \in \widehat{scc}, \hat{l}' \notin \widehat{scc}, \hat{l} \rightsquigarrow_{\hat{\varsigma}} \hat{l}'\} \\
 \hat{\varsigma}' &= \langle \hat{c}_{\hat{\varsigma}}, \hat{\sigma}_{\hat{\varsigma}} \setminus \widehat{scc}, \hat{\sigma}_{k_{\hat{\varsigma}}} \setminus \widehat{scc}, \hat{a}_{k_{\hat{\varsigma}}}, \hat{\phi}' \rangle \\
 \hat{\phi}' &= (\hat{R} \ominus \bigsqcup_{\widehat{scc}' \in \hat{T}} [\widehat{scc}' \mapsto \widehat{scc}], (\hat{\pi} \setminus \{\widehat{scc}\}) \cup \bigcup_{\hat{l} \in \widehat{scc}} \{\{\hat{l}\}\})
 \end{aligned}$$

Program Semantics. We define the abstract collecting semantics for λ_{ANF} with abstract reference counting and cycle detection using $\hat{S}_{\text{arc}++} : \text{Exp} \rightarrow \mathcal{P}(\widehat{\Sigma}_{\text{arc}++})$.

$$\hat{S}_{\text{arc}++}(e) = \{\hat{\varsigma} \in \widehat{\Sigma}_{\text{arc}++} \mid \text{inject}(e) \xrightarrow{*}_{\text{arc}++} \hat{\varsigma}\}$$

3.3.3 Properties of $(\xrightarrow{*}_{\text{arc}++})$

Performing cycle detection through a search comes at a cost, especially when looking at the theoretical worst-case behaviour. In practice, however, the advantages of being able to reclaim garbage cycles appear to outweigh this cost (cf. Section 3.4.2). Note that the backward search is only initiated when an address is being reused, as for a fresh address $\hat{l}$, there are no incoming references to traverse (i.e., $\hat{\phi}_{\hat{\varsigma}}(\hat{l}) = \emptyset$), assuming $\widehat{\mathcal{R}}(\hat{\varsigma}) = \widehat{\mathcal{S}}(\hat{\varsigma})$. As address reuse often creates a cycle, incorporating cycle detection (using `ADDEDGES` of Algorithm 1) usually pays off. In fact, for the continuation store, we can state this more formally in Theorem 7.

Theorem 7 *In a garbage-free abstract interpreter, extending abstract reference counting with cycle detection for the continuation store only requires amortised $\mathcal{O}(1)$ additional operations per continuation that is added to the continuation store.*

Proof. Note that there can never be a reference from an address $\hat{a} \in \widehat{\text{Addr}}$ to a continuation address $\hat{a}_k \in \widehat{\text{KAddr}}$. This implies that an SCC consists either exclusively of addresses in $\widehat{\text{Addr}}$ or addresses in $\widehat{\text{KAddr}}$. We write $\hat{\pi}_k$ for the partitioning $\hat{\pi}$ limited to $\widehat{\text{KAddr}}$, i.e., $\hat{\pi}_k = \hat{\pi} \setminus \mathcal{P}(\widehat{\text{Addr}})$.

We use the *potential method* for amortised analysis [156]. Assume a state $\hat{\zeta}$ which is GC sound and garbage-free (i.e., $\widehat{\mathcal{R}}(\hat{\zeta}) = \widehat{\mathcal{S}}(\hat{\zeta})$) and let $\hat{\phi}_{\hat{\zeta}} = (\hat{R}, \hat{\pi})$. We define its potential Φ as the number of references between SCCs in the continuation store $\hat{\sigma}_k$ (i.e., $\Phi(\hat{\zeta}) = \sum_{\widehat{scc}_k \in \hat{\pi}_k} |\hat{R}(\widehat{scc}_k)|$). Cycle detection is only triggered for the continuation store on a transition $\hat{\zeta} \xrightarrow{0} \hat{\zeta}'$ where we insert a new continuation $\hat{k}$ into the continuation store $\hat{\sigma}_{k\hat{\zeta}}$ (in the case of λ_{ANF} , this only happens in the transition rule [E-LET]). Since it is clear that $\Phi(\hat{\zeta}) \geq 0$, we can formulate the amortised cost $\hat{c}$ of such an insertion as $c + (\Phi(\hat{\zeta}') - \Phi(\hat{\zeta}))$, where c is the actual cost of the insertion (including cycle detection, whose cost is proportional to the traversal of the backward search). If the address $\hat{a}_{k\hat{\zeta}'}$ is fresh (i.e., $\hat{a}_{k\hat{\zeta}'} \notin \widehat{\mathcal{S}}(\hat{\zeta})$), then the cycle detection requires no traversal, hence $\hat{c} = 1 + (\Phi(\hat{\zeta}') - \Phi(\hat{\zeta}))$. We have that $\Phi(\hat{\zeta}') = \Phi(\hat{\zeta}) + 1$, since the insertion adds $\hat{a}_{k\hat{\zeta}'}$ to the set $\hat{\phi}(\hat{a}_{k\hat{\zeta}})$, which results in $\hat{c} = 2$. If the address $\hat{a}_{k\hat{\zeta}'}$ is reused (i.e., $\hat{a}_{k\hat{\zeta}'} \in \widehat{\mathcal{S}}(\hat{\zeta})$), then cycle detection will perform a backward search starting from $\hat{a}_{k\hat{\zeta}'}$. The key insight is that every address $\hat{l}$ traversed by SEARCH will lead to $\hat{a}_{k\hat{\zeta}}$, hence we are guaranteed to have a cycle. For every such $\hat{l}$, we have that $\hat{l} \xrightarrow{*}_{\hat{\zeta}} \hat{a}_{k\hat{\zeta}'}$ due to the backward search, and also that $\hat{a}_{k\hat{\zeta}} \xrightarrow{*}_{\hat{\zeta}} \hat{l}$ due to the garbage-free property of $\hat{\zeta}$ (since $\hat{l} \in \widehat{\mathcal{S}}(\hat{\zeta})$ implies $\hat{l} \in \widehat{\mathcal{R}}(\hat{\zeta})$, hence it must be that $\hat{a}_{k\hat{\zeta}} \xrightarrow{*}_{\hat{\zeta}} \hat{l}$). As a result, all references that are traversed will become part of the same SCC, and therefore be removed from $\hat{\phi}$. Hence, if cycle detection traverses k references, we have that $c = k + 1$ (due to the traversal of k references) and $\Phi(\hat{\zeta}') = \Phi(\hat{\zeta}) - k$ (due to the removal of k references), which results in $\hat{c} = c + (\Phi(\hat{\zeta}') - \Phi(\hat{\zeta})) = 1$. In both cases, the insertion only requires amortised $\mathcal{O}(1)$ operations. $\square$

The intuition here is that every reference that is inserted into $\hat{\sigma}_k$ can only be traversed once during the backward search of the cycle detection: once traversed, we can show that it is guaranteed to become part of a cycle. Since the internal references of a cycle are not traversed by the backward search, it can no longer add to the cost of a future traversal once the SCCs are merged. That is, when a backward search is triggered in the continuation store, we are guaranteed that all traversed edges between SCCs will become part of the same SCC after cycle detection, hence the path is shortened for future traversals. In practical terms, Theorem 7 states that cycle detection can be applied to the continuation store “for free”, i.e., without much of an additional cost in performance.

Note that the proof of Theorem 7 relies on the fact that λ_{ANF} does not support first-class continuations. Without first-class continuations, we can guarantee that continuation addresses ($\widehat{\text{KAddr}}$) never end up in the value store $\hat{\sigma}$, and therefore it becomes impossible to create cycles that span both the value store $\hat{\sigma}$ and the continuation store $\hat{\sigma}_k$. Because of this, every continuation address in $\widehat{\mathcal{R}}(\hat{\varsigma})$ is reachable from the current continuation address $\hat{a}_{k\hat{\varsigma}}$. Together with garbage-freeness, this ensures that every edge traversed by the backward search becomes part of a new cycle. With first-class continuations, this reasoning no longer holds, as cycles could contain both value ($\widehat{\text{Addr}}$) as well as continuation ($\widehat{\text{KAddr}}$) addresses. Consequently, the amortised $\mathcal{O}(1)$ bound would be invalid.

Importantly, as garbage cycles are now also reclaimed, we can now state that an abstract interpreter using the transition relation ($\widehat{\rightarrow}_{\text{arc}++}$) is both GC sound and garbage-free. This is stated in Theorem 8.

Theorem 8 ($\widehat{\rightarrow}_{\text{arc}++}$) is GC sound and garbage-free.

Proof. The proofs of Theorem 4 and Theorem 5 can be repeated for ($\widehat{\rightarrow}_{\text{arc}++}$), replacing addresses with their corresponding strongly connected components where necessary. The premise of Theorem 5 that disallows cycles can then be omitted, since cycles by definition do not occur between SCCs. $\square$

We can now claim full equivalence of ($\widehat{\rightarrow}_{\widehat{\Gamma}}$) and ($\widehat{\rightarrow}_{\text{arc}++}$). First, we adapt function $\widehat{rc}$ to map states from $\widehat{\Sigma}$ to the updated state space $\widehat{\Sigma}_{\text{arc}++}$.

$$\begin{aligned} \widehat{rc}(\hat{\varsigma}) &= \langle \hat{c}_{\hat{\varsigma}}, \hat{\sigma}_{\hat{\varsigma}}, \hat{\sigma}_{k\hat{\varsigma}}, \hat{a}_{k\hat{\varsigma}}, (\hat{R}_{\hat{\varsigma}}, \hat{\pi}_{\hat{\varsigma}}) \rangle \quad \textbf{where} \\ \hat{\pi}_{\hat{\varsigma}} &= \{ \{ \hat{l}' \in \widehat{\text{Loc}} \mid \hat{l} \overset{*}{\rightsquigarrow}_{\hat{\varsigma}} \hat{l}' \wedge \hat{l}' \overset{*}{\rightsquigarrow}_{\hat{\varsigma}} \hat{l} \} \mid \hat{l} \in \widehat{\text{Loc}} \} \\ \hat{R}_{\hat{\varsigma}}(\widehat{scc}) &= \{ \hat{l}' \in \widehat{\text{Loc}} \mid \hat{l}' \notin \widehat{scc} \wedge \exists \hat{l} \in \widehat{scc}, \hat{l}' \rightsquigarrow_{\hat{\varsigma}} \hat{l} \} \quad \textbf{for } \widehat{scc} \in \hat{\pi}_{\hat{\varsigma}} \end{aligned}$$

Theorem 9 The transition relation ($\widehat{\rightarrow}_{\text{arc}++}$) is equivalent to ($\widehat{\rightarrow}_{\widehat{\Gamma}}$), in the sense that: $\forall \hat{\varsigma}, \hat{\varsigma}' \in \widehat{\Sigma}, \hat{\varsigma} \widehat{\rightarrow}_{\widehat{\Gamma}} \hat{\varsigma}' \iff \widehat{rc}(\hat{\varsigma}) \widehat{\rightarrow}_{\text{arc}++} \widehat{rc}(\hat{\varsigma}')$.

Proof. The proof is analogous to that of Theorem 6. $\square$

Corollary 4 $\forall e. \hat{S}_{\text{arc}++}(e) = \{ \widehat{rc}(\hat{\varsigma}) \mid \hat{\varsigma} \in \hat{S}(e) \}$.

Proof. Follows directly from Theorem 9. $\square$

3.4 Evaluation

We have already proven the GC soundness and garbage-freeness of our approach (Theorem 8), as well as a theoretical efficiency claim of the cycle detection technique for the continuation store (Theorem 7). We now present the empirical evaluation, for which we formulate two main research questions:

- RQ1** What is the impact of the cycle detection technique presented in Section 3.3.2, both in terms of overhead as well as precision gains?
- RQ2** How does abstract reference counting compare (in terms of precision and performance) to existing application policies for abstract tracing GC?

Section 3.4.2 measures the impact of the cycle detection technique on abstract reference counting, while Section 3.4.3 compares its precision and performance to existing approaches to abstract GC.

3.4.1 Experimental Setup

We use a similar setup throughout our evaluation.

Environment. We ran all experiments using Scala version 2.12.3 on a server with a 3.5GHz Intel Xeon E5-2637 processor and 256GB of RAM.

Implementation. We implemented both abstract reference counting and the existing variants of abstract tracing GC (surveyed in Section 3.1.1) using the Scala-AM framework for abstract interpretation of Scheme programs [152, 151]. This resulted in six abstract interpreters⁹, each incorporating the Scheme equivalent of the λ_{ANF} transition relations ($\hat{\rightarrow}$) (which does not perform any GC) and ($\hat{\rightarrow}_{\hat{f}}$) (which performs GC at every step) from Section 2.3, ($\hat{\rightarrow}_{\text{arc}}$) (which performs abstract reference counting without cycle detection) from Section 3.2, ($\hat{\rightarrow}_{\text{arc}++}$) (which performs abstract reference counting with cycle detection) from Section 3.3, as well as the additional ($\hat{\rightarrow}_{\text{FCFA}}$) and ($\hat{\rightarrow}_{\text{arc}+}$) relations defined below:

- ($\hat{\rightarrow}_{\text{FCFA}}$) adopts the GC policy of FCFA [106], which applies abstract GC whenever values need to be joined in the store, so that values are by design never merged with garbage. While this approach is not garbage-free (Section 3.1.1), it is interesting to examine how this common policy affects the precision and performance of the abstract interpreter.
- ($\hat{\rightarrow}_{\text{arc}+}$) adopts abstract reference counting, but only applies cycle detection to the continuation store $\hat{\sigma}_k$. As garbage cycles will remain in the value

⁹Publicly available at <https://github.com/noahvanes/scala-am-abstractgc>.

store $\hat{\sigma}$, this relation is not garbage-free. However, its empirical evaluation is motivated by our observation that garbage cycles are mainly an issue in the continuation store, and by our proof for Theorem 7 that cycle detection should have no major performance overhead there.

All interpreters support strong updates through abstract counting.

Benchmarks. We evaluate the above implementations using a total of 16 Scheme programs; 11 from the Gabriel benchmark suite [49] and 5 from the built-in test suite of Scala-AM. We included all programs from the Gabriel benchmark suite that were compatible with Scala-AM¹⁰, and selected benchmarks from the built-in test suite of Scala-AM that could be analysed within a 30 minute timeout using $(\Rightarrow)$ while producing at least 500 states (so that we can measure meaningful precision benefits). Table 3.1 lists the size of each Scheme program, with the Gabriel ones depicted in **bold**. For each benchmark program, we configure each abstract interpreter with a monovariant allocator (i.e., OCFA), and a lattice that abstracts concrete values to the set of their possible types (a finite domain, so that termination is guaranteed, even with abstract GC and strong updates).

Benchmark	LOC	Benchmark	LOC	Benchmark	LOC	Benchmark	LOC
boyer	568	deriv	39	takl	18	gcipd	12
browse	78	destruc	37	puzzle	144	printrtest	35
cpstak	19	diviter	8	triangl	40	rsa	41
dderiv	83	divrec	11	collatz	22	nqueens	34

Table 3.1: Lines of code per benchmark. The Gabriel benchmarks are in **bold**.

Measurements. As the implementations differ only in their approach to abstract garbage collection, we follow Earl et al. [45] in the use of the number of explored states as a proxy for *precision* measurements. As garbage can cause an abstract interpreter to explore spurious states (cf. Section 3.1.1), fewer states implies higher precision when all configuration parameters are kept constant. For *performance*, we measure the total running time of the abstract interpreter. Note that running time is impacted by two factors. First, the number of states the interpreter has to explore. Higher precision can often lead to higher performance. Second, the rate at which the abstract interpreter is able to explore those states. This rate is influenced by the *overhead* caused by abstract garbage collection, which we therefore also measure per state. For the tracing GC approaches ($(\Rightarrow_{\hat{F}})$ and $(\Rightarrow_{\text{GCFA}})$), this

¹⁰We omitted the *ctak* benchmark program due to its use of `call/cc`, which is not supported by Scala-AM. This is a limitation of the implementation, not of abstract reference counting.

measure corresponds to the time spent in the function $\hat{\Gamma}$. For approaches based on abstract reference counting ($(\Rightarrow_{\text{arc}})$, $(\Rightarrow_{\text{arc}+})$ and $(\Rightarrow_{\text{arc}++})$), this measure includes the time spent on updating the $\hat{\phi}$ component (including cycle detection for $(\Rightarrow_{\text{arc}+})$ and $(\Rightarrow_{\text{arc}++})$) and in the $\widehat{\text{collect}}$ function. We ran every benchmark 30 times after a warm-up period of 2 minutes. For the time-sensitive measurements, we report the mean of the measurements of all runs.

3.4.2 RQ1: Impact of Cycle Detection

Section 3.2 introduced transition relation $(\Rightarrow_{\text{arc}})$ which performs abstract reference counting, but cannot reclaim cycles. Section 3.3 extended this relation into $(\Rightarrow_{\text{arc}++})$, which can reclaim cyclic garbage by applying cycle detection to both stores. Transition relation $(\Rightarrow_{\text{arc}+})$, introduced in Section 3.4.1, is an in-between: it uses abstract reference counting, but only applies cycle detection to the continuation store. Table 3.2 compares the corresponding abstract interpreters to evaluate the impact of cycle detection on abstract reference counting.

Impact on Precision. The results from Table 3.2 show that reclaiming garbage cycles from the continuation store only can already result in important precision improvements compared to naive abstract reference counting with $(\Rightarrow_{\text{arc}})$. For most benchmarks, $(\Rightarrow_{\text{arc}+})$ nearly achieves the same optimal precision as $(\Rightarrow_{\text{arc}++})$. For instance, all of the non-Gabriel benchmarks (except for *collatz*) show significantly improved precision with cycle detection for the continuation store only, and adding cycle detection to the value store does not increase precision any further (save for a small improvement in *nqueens*). In addition, for the *triangl* benchmark, not reclaiming garbage cycles from the continuation store (using $(\Rightarrow_{\text{arc}})$) causes the state space to blow up. However, it is clear that reclaiming them from the continuation store does not always suffice: benchmarks *destruc* and *puzzle* both create cycles in the value store, which are not reclaimed by neither $(\Rightarrow_{\text{arc}})$ nor $(\Rightarrow_{\text{arc}+})$, therefore only showing significantly improved precision with $(\Rightarrow_{\text{arc}++})$. Programs that are written in CPS, such as *cpstak*, do not benefit from the extra precision in the continuation store either, as their control flow is encoded in closures that reside in the value store. The garbage-free transition relation $(\Rightarrow_{\text{arc}++})$ improves precision more consistently, due to its reclamation of all garbage cycles from both stores. These results underline that cycles cannot be neglected in abstract reference counting, and that a technique for their detection is required to realise the benefits of garbage-free abstract interpretation.

Impact on Performance. Figure 3.6 compares the time spent on GC per computed state for the reference-counting transition relations. For most benchmarks,

	$(\hat{\rightarrow}_{\text{arc}})$		$(\hat{\rightarrow}_{\text{arc}+})$		$(\hat{\rightarrow}_{\text{arc}++})$	
	t	#s	t	#s	t	#s
cpstak	3	179	3	179	3	165
diviter	15	163	9	163	19	162
divrec	10	153	15	146	16	145
destruc	137 384	1 874 444	142 272	1 874 444	825	48 006
triangl	∞	>1 973 282	1 745	2 734	1 981	2 696
puzzle	∞	>15 099 494	∞	>15 099 494	2 623	109 420
takl	11 730	862 880	2 039	188 554	2 705	188 551
browse	∞	>3 475 125	∞	>2 321 534	∞	>2 309 698
boyer	∞	>13 965 902	∞	>11 771 411	∞	>10 363 208
deriv	∞	>13 624 193	∞	>30 991 292	∞	>27 055 217
dderiv	∞	>25 051 847	∞	>35 209 950	∞	>33 565 445
collatz	<1	60	<1	60	<1	60
gcipd	<1	151	<1	91	<1	91
printest	94	9 803	11	1 537	11	1 537
rsa	92	10 646	12	1 538	14	1 538
nqueens	3 777	146 525	540	40 401	769	40 375

Table 3.2: Comparison of time taken in milliseconds (t; lower means better performance) and number of states explored (#s; lower means better precision). A time of ∞ means that the benchmark exceeded the time limit of 30 minutes; in this case, we report the number of states explored when the timeout was reached.

we observe that cycle detection increases the GC overhead slightly. However, this overhead is negligible compared to the corresponding improvements to precision. That is, the overheads in Figure 3.6 remain comparable for different configurations on each benchmark, while the number of states explored reported in Table 3.2 decreases significantly when using cycle detection. We observe proportionally large improvements to performance, as the abstract interpreter has fewer states to explore. Again, the garbage-free transition relation $(\hat{\rightarrow}_{\text{arc}++})$ can most consistently improve performance by completely avoiding any unnecessary computation, with only a negligible increase in overhead. We conclude that the benefits of cycle detection outweigh its cost, as $(\hat{\rightarrow}_{\text{arc}++})$ is superior to both $(\hat{\rightarrow}_{\text{arc}})$ and $(\hat{\rightarrow}_{\text{arc}+})$ in terms of precision and performance.

Note that Table 3.2 also lists the number of states that were explored for the benchmarks that did time out. This gives a rough estimate of the interpreters' state exploration rate. However, these rates should be interpreted with caution, as some paths in the state space can be significantly cheaper to explore than others. One should therefore only compare the rates of interpreters with the same precision, or look for a pattern of major discrepancies. We do not discern such a pattern among the rates, and deem the differences in precision too pronounced to warrant a direct rate comparison.

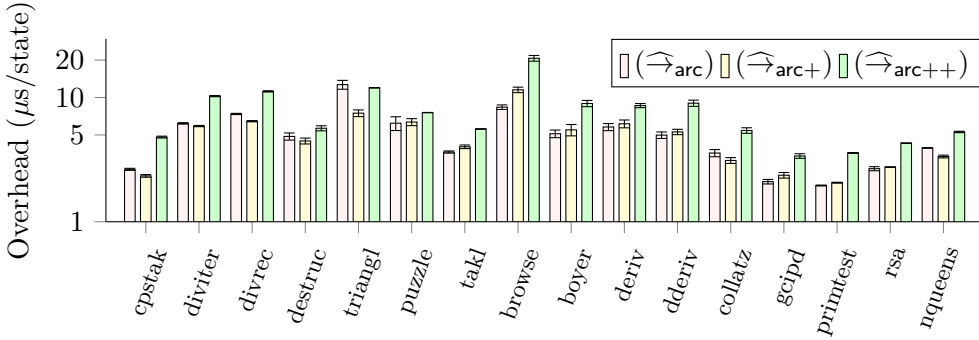


Figure 3.6: Comparison of overhead measured in time spent on GC per state (lower is better). Error bars indicate the standard deviation of the mean of our measurements (for 30 runs). Note the usage of a logarithmic scale on the y-axis.

3.4.3 RQ2: Comparison to Existing Policies for Abstract GC

Table 3.3 compares $(\Rightarrow_{\text{arc}++})$ to other approaches to abstract GC. On one end of the spectrum, there is $(\Rightarrow)$, which never applies abstract GC. On the other end, there is $(\Rightarrow_{\text{f}})$, which applies abstract GC on every evaluation step. An interesting tradeoff is made by $(\Rightarrow_{\text{RCFA}})$, which applies GC before every store join.

	$(\Rightarrow)$		$(\Rightarrow_{\text{f}})$		$(\Rightarrow_{\text{RCFA}})$		$(\Rightarrow_{\text{arc}++})$	
	t	#s	t	#s	t	#s	t	#s
cpstak	6	355	30	165	14	179	3	165
diviter	23	175	44	162	28	163	19	162
divrec	14	219	30	145	23	153	16	145
destruc	233 148	8 336 390	10 468	48 006	4 253	63 235	825	48 006
trianagl	∞	>4 165 359	2 157	2 696	2 248	2 905	1 981	2 696
puzzle	∞	>83 640 660	21 895	109 420	26 212	374 231	2 623	109 420
takl	123 829	7 077 844	36 352	188 551	22 163	255 763	2 705	188 551
browse	∞	>5 597 675	∞	>1 605 720	∞	>1 891 118	∞	>2 309 698
boyer	∞	>6 502 235	∞	>316 535	∞	>1 696 189	∞	>10 363 208
deriv	∞	>78 223 109	∞	>6 378 232	∞	>11 565 814	∞	>27 055 217
dderiv	∞	>66 122 960	∞	>4 444 925	∞	>12 020 387	∞	>33 565 445
collatz	5	607	5	60	6	159	<1	60
gcipd	9	1 098	9	91	5	147	<1	91
printest	15 047	1 084 531	204	1 537	173	3 622	11	1 537
rsa	10 544	625 795	211	1 538	145	2 793	14	1 538
nqueens	791 837	48 699 676	8 859	40 375	5 410	59 717	769	40 375

Table 3.3: Comparison of time taken in milliseconds (t; lower means better performance) and number of states explored (#s; lower means better precision) by an abstract interpreter for different approaches to abstract GC. A time of ∞ means that the benchmark exceeded the time limit of 30 minutes; in this case, we report the number of states explored when the timeout was reached.

Comparison of Precision. Our results confirm the findings of previous work [106, 107, 45, 59]: abstract GC reduces the state space to explore substantially. Looking at the number of explored states in Table 3.3 reveals that the interpreter without abstract GC using $(\hat{\Rightarrow})$ has the lowest precision for every benchmark compared to all other approaches. The interpreter using $(\hat{\Rightarrow}_{\Gamma\text{CFA}})$ sacrifices some precision by invoking abstract GC less frequently than $(\hat{\Rightarrow}_{\hat{\Gamma}})$. It explores more states than necessary because it is not garbage-free, which is the most noticeable on more complex programs such as *destruc*, *puzzle*, *takl* and *nqueens*.

Our approach, $(\hat{\Rightarrow}_{\text{arc}++})$, achieves the same optimal precision as $(\hat{\Rightarrow}_{\hat{\Gamma}})$. As we have proven that both transition relations are garbage-free and equivalent (cf. Theorem 9), they explore the same number of states.

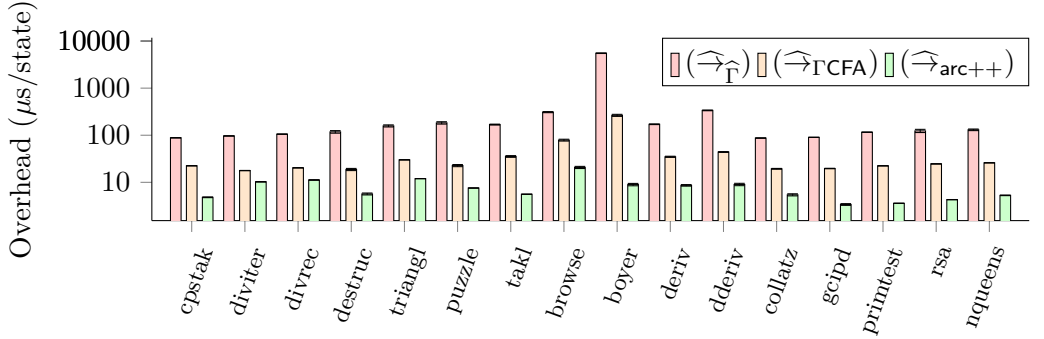


Figure 3.7: Comparison of overhead measured in time spent on GC per state (lower is better). Error bars indicate the standard deviation of the mean of our measurements (for 30 runs). Note the usage of a logarithmic scale on the y-axis.

Comparison of Performance. Figure 3.7 compares the overhead of abstract GC for $(\hat{\Rightarrow}_{\hat{\Gamma}})$, $(\hat{\Rightarrow}_{\Gamma\text{CFA}})$ and $(\hat{\Rightarrow}_{\text{arc}++})$. The $(\hat{\Rightarrow})$ interpreter does not suffer such overhead. However, without any abstract GC, its state space becomes polluted (leading to repeated exploration of equivalent states) and explodes (due to infeasible paths). It exhibits the worst performance.

The overhead of abstract GC is an issue for $(\hat{\Rightarrow}_{\hat{\Gamma}})$. It is clear that a tracing GC application is generally more expensive than a single evaluation step. Applying tracing GC after every step leads to the interpreter’s running time being dominated by that of the GC applications. This is pronounced on complex benchmarks that require larger heaps such as *boyer*: since tracing GC needs to traverse the entire heap, the overhead grows with the size of the heap. For the *boyer* benchmark, the interpreter using $(\hat{\Rightarrow}_{\text{arc}++})$, which explores the same state space as $(\hat{\Rightarrow}_{\hat{\Gamma}})$ for

all benchmarks, explores 10.4M states before timeout, whereas with the excessive GC overhead of $(\Rightarrow_{\hat{f}})$, it can only explore 317K states with the same time limit. While the garbage-freeness of $(\Rightarrow_{\hat{f}})$ significantly improves overall performance of $(\Rightarrow)$ by reducing the state space, it is not as fast as $(\Rightarrow_{\text{arc}++})$ due to the higher overhead. In Figure 3.7, for all benchmarks the overhead of $(\Rightarrow_{\hat{f}})$ differs from that of $(\Rightarrow_{\text{arc}++})$ by orders of magnitude, which results in significant performance improvements in Table 3.3 while exploring the same number of states. The transition relation $(\Rightarrow_{\text{GCFA}})$ avoids this overhead of $(\Rightarrow_{\hat{f}})$ by applying abstract GC less frequently. As such, its overhead is significantly lower than that of $(\Rightarrow_{\hat{f}})$. For nearly all our benchmarks, this results in a performance improvement over $(\Rightarrow_{\hat{f}})$, despite having to explore a larger state space. Note that $(\Rightarrow_{\text{GCFA}})$ is no longer garbage-free and explores more states than necessary. For instance, on the *puzzle* benchmark it explores more than 3 times as many states as $(\Rightarrow_{\text{arc}++})$. In addition, Figure 3.7 reveals that the overhead remains considerably higher than that of $(\Rightarrow_{\text{arc}++})$. This can also be seen in the benchmarks that time out, where the difference in states explored is significant compared to $(\Rightarrow)$ and even $(\Rightarrow_{\text{arc}++})$. In particular, the *boyer* benchmark also shows that $(\Rightarrow_{\text{GCFA}})$ still scales poorly towards larger heap sizes, an inherent downside of tracing GC (due to the need to traverse the entire heap). Since both overhead and number of states explored are higher than for $(\Rightarrow_{\text{arc}++})$, the overall performance is worse.

In general, when determining an application policy for abstract tracing GC, sacrificing garbage-freeness for increased performance implies lower precision. Using abstract reference counting with $(\Rightarrow_{\text{arc}++})$ gives the best of both worlds. Its low overhead is not detrimental to performance: although there is definitely some overhead compared to $(\Rightarrow)$, it is consistently lower than all other approaches that actually apply abstract GC. Compared to the equivalent transition relation $(\Rightarrow_{\hat{f}})$ with optimal precision, overhead is reduced by orders of magnitude. The overall performance of the *triangl* benchmark is an exception to the rule: even though the GC overhead is greatly reduced when using $(\Rightarrow_{\text{arc}++})$, its running time is mostly dominated by expensive evaluation steps, but even in this case none of the other interpreters outperform $(\Rightarrow_{\text{arc}++})$. Since $(\Rightarrow_{\text{arc}++})$ achieves the optimal precision, overall performance is not impacted by the exploration of spurious states. Hence, our experiments show that $(\Rightarrow_{\text{arc}++})$ is superior both in terms of precision and performance, as it achieves the best result in both for all benchmarks.

3.5 Related Work

The focus of this chapter is a more efficient approach to abstract GC. It relates to previous work on both concrete and abstract GC, discussed further below.

Reference Counting. Our approach is based on the original formulation of reference counting [30], without any common optimisations [144] such as deferred [41] and coalesced [94] reference counting or “lazy freeing” [21]. These optimisations postpone the reclamation of garbage, which decreases overhead for a concrete interpreter, but inhibits garbage-freeness for an abstract interpreter.

We perform abstract reference counting at the level of SCCs, so that garbage cycles can be reclaimed. For concrete reference counting, two major techniques have been proposed to handle cycles: using a back-up tracing garbage collector [179] and trial deletion [29]. We deem both too expensive for frequent applications during abstract interpretation, which is required to maintain garbage-freeness. Nevertheless, it could be interesting to investigate a variant of abstract reference counting where abstract tracing GC is applied at the evaluation steps that might render a cycle into a garbage cycle. In the concrete, this has been shown [98] to occur when a memory location loses some, but not all of its incoming references. Alternatively, a pre-analysis as in [10] could identify the structures guaranteed to be acyclic beforehand. Concretely, for our approach, this could avoid the need to apply cycle detection (Algorithms 1 and 2) for such structures, potentially further reducing the overhead of $(\widehat{\rightarrow}_{\text{arc}++})$, if the pre-analysis itself is cheap enough.

Confusingly, the term “abstract reference counting” has been used with different meanings. Hudak [68] formalises reference counting for a concrete interpreter. After abstraction, he obtains a static analysis that approximates the *concrete* reference count, enabling program optimisations [69, 116] and compile-time GC [79]. The analysis exemplifies static analysis *of* reference counting, rather than static analysis *with* reference counting for the purpose of improving precision and performance. Finally, Jones et al. [78] use the term to abstract over the low-level details concrete implementations differ on.

Tracing Garbage Collection. The main shortcoming of tracing GC is that it requires frequent heap traversals to keep the abstract interpreter garbage-free. Various techniques exist for concrete interpreters to avoid the correspondingly long GC pauses and to improve their throughput. Incremental algorithms, such as Baker’s incremental stop-and-copy algorithm [12], only remove garbage lazily, and can therefore not be used in our setting. Moreover, they would need to be redesigned for abstract interpreters which do not always allocate a fresh address. Generational algorithms [96] can increase throughput by limiting heap traversals to the young generations only. However, as garbage also arises in older ones, it does not suffice to keep the abstract interpreter garbage-free. A full GC would be required to prevent precision losses for reallocated addresses in the old generation.

Abstract Garbage Collection. The Γ CFA analysis [106, 103, 107] pioneered the concept of abstract garbage collection for the analysis of dynamic, higher-order languages using AAM-like abstract interpreters. The term “garbage-free” was later coined in [168] to describe an abstract interpreter that applies abstract GC at every evaluation step, eliminating all garbage. Similar to our own work, Γ CFA also incorporated abstract counting to allow for strong updates, as this technique synergises well with abstract GC (cf. Section 2.3.3).

Notably, both abstract GC and abstract counting were already used in earlier work by Jagannathan et al. [71] for the analysis of higher-order languages. Their analysis computes reachability for each expression (analogous to our function $\hat{R}$) *a priori*; however, the computation of this reachability information in turn requires the analysis results (specifically, it requires the store computed by the analysis). Hence, they first compute an analysis with the most over-approximative (but sound) reachability information, and then iteratively recompute the analysis with more precise reachability information, computed using the results of the previous analysis, repeating this iteration until a fixpoint is reached.¹¹ In contrast, other work using abstract GC (including our own) typically computes such reachability information “on-the-fly” during the analysis itself.

We discuss related work on abstract GC for pushdown CFAs in Section 4.7.

Other Forms of Abstract GC. Different forms of abstract GC have also been commonly used in other dataflow analyses [121], where the technique is often better known as *memory localisation*, *context projection* or *framing*. The main motivation for these techniques is that they can be used to efficiently decompose the analysis of larger programs into the local analyses of smaller components. These local analyses produce summaries that are reusable, as they are restricted to the memory footprint that each component operates over. As such, abstract GC can be considered an enabler for *local reasoning* and *compositional analysis*.

Context projection was introduced by Chen et al. [27] as a technique to limit (i.e., to “project”) the input state (i.e., the “context”) in which a node is evaluated to only the parts that were relevant for that node. In this setting, the primary goal of context projection was not to improve precision, but to decrease the memory consumption of the analysis and to avoid redundant evaluations for states that differ only in parts that are not relevant. We measured the impact of such redundant evaluations as well: Γ CFA by design does not lose precision due to joins with garbage, but still generates more states than necessary, as it explores states that are identical modulo unreachable store bindings multiple times.

¹¹One could also stop this iteration before a fixpoint is reached, which would yield a sound, but over-approximative result.

Marron et al. [97] perform shape analysis using graph-based models of the heap, incorporating context projection (adapted for these graph models) to improve both analysis performance *and* precision. They point out that in this setting, applying context projection not only improves reuse of analysis results (and hence analysis performance), but also avoids precision loss for “recursive procedures that use a summary representation for multiple out-of-scope references”. The latter corresponds to the main motivation for our own use of abstract GC — that is, to avoid the precision loss from joining with out-of-scope (i.e., “garbage”) values.

Other work on shape analysis [135, 60, 125, 182, 70] uses *framing* as a well-known technique to improve reuse of previously computed analysis results. Such analyses are often expressed using separation logic [133], an extension of Hoare logic that uses assertions to reason over the layout and contents of (an abstraction of) the heap. At the core of separation logic is the FRAME rule, which can be seen as a form of abstract GC. The FRAME rule limits the pre- and post-condition of a Hoare triple for a statement s to only the part of the heap that is accessed by s .

$$\frac{\{A\} \ s \ \{B\}}{\{A * F\} \ s \ \{B * F\}} \text{ [FRAME]}$$

That is, this rule allows “temporarily” collecting unreachable bindings in F , so that a more localised Hoare triple for s can be reused. By reducing the Hoare triples to a minimal summary, analysis results can be reused more often in different contexts with similar sharing patterns, greatly increasing the analysis’ scalability [135]. The FRAME rule crucially makes use of the separating conjunction ($*$) instead of the regular logic conjunction ($\wedge$) to enforce that the excluded “garbage” (i.e., F) does not overlap with the part of the heap that is accessed. Similar to the tracing GC function $\hat{\Gamma}$, the FRAME rule can be applied at any point, but in practice is most often applied at a function entry (e.g., as is done in the “local interprocedural analysis” of [60]). In order to apply the FRAME rule automatically, frame inference [15] splits the caller’s symbolic heap into the portion required by the callee and the leftover portion (“the frame”) that is not accessed by the callee. Bi-abduction [25] extends frame inference with abduction by inferring both the frame and the *anti-frame* (containing assertions required by the callee but missing from the caller’s heap), from which a procedure’s precondition is automatically derived. The combination of the FRAME rule and bi-abduction has been the foundation of the success of Facebook’s Infer tool [26]: bi-abduction ensures that the Hoare triples of procedures can be derived independently of their callers, while the FRAME rule enables reusing these local summaries in different call contexts.

Alternatives to Reachability. As pointed out by Chen et al. [27], abstract GC need not necessarily be defined in terms of reachability (although they only present a reachability-based approach as an example to abstract GC themselves).

For instance, Oh et al. [122] eliminate store bindings in their analysis based on which addresses can be used, guided by a pre-analysis to conservatively approximate which bindings are never used and can safely be removed as garbage. This is in contrast to our own approach to abstract GC, which by design is purely based on reachability. Note that in practice, we enjoy some of the same precision gains presented by the authors of this work by simply limiting environments (such as those captured by closures) to include only free variables, resulting in a smaller root set and therefore a more aggressive form of abstract GC. We have elided this optimisation from our formalisation for brevity, but integrated it in the implementation of all abstract interpreters in Scala-AM.

Might et al. [108] proposed a similar extension to abstract GC, named *conditional* abstract GC, which only keeps bindings in the store that are reachable *and* satisfy certain conditions. These conditions (referred to as *governors*) are syntactically derived from conditional expressions (e.g., the predicate of an if-expression). The technique then relies on a theorem prover to determine (and discard) bindings that would only be used under unsatisfiable conditions. However, to the best of our knowledge, this form of conditional abstract GC has not yet been implemented in practice.

Jensen et al. [74] propose *lazy propagation* to tackle some of the same issues as abstract GC. The idea behind lazy propagation is to avoid immediately passing all heap entries to a called function; instead, heap entries are initialised to a special **unknown** value, and the real value is only looked up through a special recovery operation when it is first accessed. As such, non-garbage bindings are determined according to what is demanded, not what is reachable. The main benefit of lazy propagation is that — similar to abstract GC — redundant re-analyses due to irrelevant differences in garbage bindings are avoided. A second benefit is that lazy propagation can also increase precision: heap entries that remained untouched (i.e., are still bound to **unknown**) upon return are restored to their original binding in the caller's store, therefore skipping any joins in the callee with values from other callers. Abstract GC, as discussed in this chapter (and Chapter 2), is not able to perform this precision optimisation; in Chapter 4, we introduce a form of *stackless GC* [53] that achieves this by temporarily collecting such bindings. Note, however, that lazy propagation never permanently collects garbage bindings, and therefore misses out on the core precision benefits of abstract GC.

Lazy propagation is used in the TAJJS analysis framework for JavaScript [73, 74], which also applies abstract GC at every function exit¹². Due to lazy propagation, abstract GC in TAJJS does not have access to all references of the callers of the current function invocation (and therefore, the full GC root set), and cannot directly traverse the entire heap. Therefore, it conservatively treats every binding that may have already existed upon the entry of the function (that is returned to) as part of the GC root set. Abstract GC then only collects bindings that are guaranteed to have been allocated during the current function invocation. Interestingly, this also results in a more lightweight form of abstract GC, as it only needs to traverse the part of the heap that is explicitly represented (i.e., not **unknown**) for the current function invocation. The downside is that because of this, not all garbage bindings can be eliminated immediately (more precisely, they can only be reclaimed when returning to a function invocation during which they were allocated). Therefore, this form of GC does not achieve garbage-freeness in general (regardless of how often or when abstract GC is applied), and does not avoid as many precision losses as a full tracing GC. Such a precision loss can occur when a new binding is allocated at the same allocation site as a garbage binding (allocated before the current function invocation) that was not yet collected.

Alternatives for Improved Efficiency. We have surveyed related work for different application policies of tracing abstract GC that aim to balance its benefits with its overhead (cf. Section 3.1.1). Similar to our own work, other efforts have also been proposed as alternative strategies to reduce this overhead.

In Knauel’s abstract interpretation framework [88], the overhead of tracing GC proved performance-detrimental on complex programs. The proposed solution is a *global garbage collector* which collects garbage for multiple states at once with a shared, global root set. Of course, this sacrifices many of the precision benefits of abstract GC, as the larger root set results in fewer bindings that can be collected.

For the analysis of Python programs, MOPSA [112] uses an alternative form of abstract GC, which the authors describe as a form of tracing GC. Their implementation, however, differs from that of a traditional tracing GC algorithm: instead of computing the set of all transitively reachable addresses from some root set, the GC algorithm of MOPSA computes the set of all addresses still reachable from another abstract value in the heap (subsequently reclaiming all addresses not in this set). This approach does not immediately collect all garbage, as addresses reachable from garbage values are also included in this set. However, repeated

¹²Neither [73] nor [74] mentions when abstract GC is applied. Instead, we extracted the GC application policy of TAJJS directly from its current implementation (<https://github.com/cs-au-dk/TAJS>, commit 3bdf55a), in which lazy propagation is enabled by default.

applications of the garbage collector can remove more garbage (i.e., their GC algorithm is not idempotent), as addresses reachable only from addresses that have been reclaimed in a previous application of GC are collected in the next application. Abstract GC in MOPSA therefore shares similarities with abstract reference counting: addresses are reclaimed when they are no longer referenced by other addresses in the heap (i.e., when their reference count drops to 0). Unlike reference counting, MOPSA’s GC algorithm still requires heap traversals and frequent applications to reclaim all (non-cyclic) garbage (whereas abstract reference counting immediately propagates decreasing reference counts as addresses are reclaimed). Similar to naive abstract reference counting (without cycle detection), abstract GC in MOPSA also cannot reclaim garbage cycles.

To our knowledge, no prior work has employed abstract reference counting as a more efficient approach to garbage-free abstract interpretation, despite reference counting being mentioned as future work in [106]. Bergstrom et al. [16, 17] employ a primitive form of reference counting as a cheaper alternative to full abstract GC. Their control flow analysis (CFA) (which does not make use of an explicit store component) tracks for every variable if it has been captured by the lexical environment of another closure; if not, this variable can safely be collected once it goes out of scope. Compared to our approach, this corresponds to abstract reference counting with reference counts from $\hat{\mathbb{N}} = \{0, 1, \infty\}$. We have shown that such an approximation is unnecessary, as the precise reference count can be maintained without having to increase the complexity of the state space.

Alternative Abstract Interpreter Designs. We present abstract reference counting in a minimal setting using the AAM approach to abstract interpretation proposed by Might et al. [168, 105]. Therefore, we elide common optimisations to AAM (such as those proposed by Glaze et al. [58]) and do not consider any of its variations (e.g., Abstracting Abstract Control (AAC) [57], P4F [56]), although our approach remains applicable in such settings as well. An advantage of AAM is that it is a systematic method for the design of a wide variety of analyses, which is applicable to programs with highly dynamic behaviour (e.g., higher-order programs with mutable state). The performance and precision improvements brought by our work should therefore carry over to such analyses. Moreover, we argue that the concepts behind abstract reference counting are sufficiently general to support its incorporation in other existing analyses that already make use of abstract tracing GC.

We discuss related work on the integration of abstract GC with pushdown analysis [45, 59], as well as the integration of store widening, later in Chapter 4.

Incremental Cycle Detection. To reclaim garbage cycles, our approach detects cycles as references are added to and removed from the store. We implement such *incremental cycle detection* using a simple backward search, which has sub-optimal worst-case performance in theory. More advanced algorithms have been proposed [136, 61, 14] with better asymptotic performance. Note that such algorithms need to support both the insertion (upon extending the store) as well as the deletion (upon strong updates and GC) of edges in the graph, which often worsens theoretical performance bounds. Empirical evidence also suggests [148] that more complicated algorithms do not always perform better in practice. Indeed, in Section 3.3 we argued that backward searching usually works well for an abstract interpreter, and we verified this empirically in Section 3.4 on our benchmark suite. Nevertheless, it would be interesting to explore whether other, more efficient incremental cycle detection algorithms can be used here instead.

3.6 Conclusion

We have introduced abstract reference counting as a more efficient approach to abstract garbage collection. Existing approaches are based on tracing GC, which is non-continuous in nature, and therefore need to trade off either precision or performance to strike the balance between the benefits and overhead of abstract GC. Our approach based on reference counting, in contrast, requires but minor book-keeping. It results in a garbage-free abstract interpreter that is as precise as one that applies tracing GC at every step — in the absence of cycles.

However, we showed that cycles are a major threat to the garbage-freeness of abstract reference counting. Next to those arising under concrete interpretation, the address allocator of an abstract interpreter can create many artificial cycles. We proposed a cycle detection algorithm as a solution, maintaining the reference counts at the level of the strongly connected components in the stores. This enables the reclamation of garbage cycles, which is necessary to make abstract reference counting garbage-free so that it can achieve the optimal precision.

In terms of precision, our experiments confirm our claim on the garbage-freeness of the abstract interpreter, showing that it achieves the optimal precision for abstract garbage collection. In terms of performance, abstract reference counting — even with cycle detection — greatly reduces overhead relative to tracing GC by avoiding frequent heap traversals. Without sacrificing any precision, it can therefore realise significant performance improvements for an abstract interpreter.

4 Delta Store Semantics

Both the precision and performance of abstract interpreters can be greatly improved through the integration of abstract garbage collection. In Chapter 3, we integrated abstract GC into a small-step “AAM-style” abstract interpreter with local stores. Unfortunately, abstract interpreters with such designs often face inherent limitations to scalability. Abstract GC does not immediately appear compatible with big-step “pushdown-style” abstract interpreters (or definitional interpreters), as these do not immediately allow recovering all references from the stack, which are needed as part of the GC root set. However, such big-step abstract definitional interpreters are often preferable over their small-step counterparts, as they offer important advantages in terms of both precision and performance. Attempts to integrate abstract GC into these interpreters either fail to deliver the optimal precision and performance benefits of both abstract GC and pushdown-style control flow, and/or require complicated modifications to the abstract interpreter. In addition, the lack of global store widening, which is incompatible with abstract GC, further impedes scalability.

In this chapter, we present delta store semantics (DSS), offering a novel yet simple approach to integrate abstract GC into big-step abstract interpreters that fully exploits “the best of both worlds” in terms of analysis precision and performance. DSS makes a simple change to the standard big-step language semantics, returning a delta store (representing changes to the original store) instead of an updated store. Because of this change, we can augment DSS with a single evaluation rule to interleave abstract GC into its semantics. Importantly, we show that DSS not only preserves the advantages of big-step abstract interpreters and abstract GC, but in fact can exploit greater precision benefits with abstract GC compared to an equivalent small-step abstract interpreter with abstract GC (such as those of Chapter 3). We formulate this claim as a theorem, for which we provide both a mechanised proof in Rocq (for programs without mutable variables), as well as empirical evidence. Finally, we propose a new form of store widening for DSS, which tackles the scalability issues of abstract interpreters employing abstract GC without store widening. The result is similar to the traditional notion of flow sensitivity in data flow analyses.

4.1 Motivation

The AAM-based abstract interpreters of Chapter 2 and Chapter 3 are designed as abstractions of the small-step semantics of λ_{ANF} . These operational semantics directly model (an abstraction of) the execution of an evaluation using a continuation-passing style (CPS), and thus also require an explicit abstraction of the continuation component of the interpreter (i.e., “the (control) stack”).

In contrast, an abstract interpreter can also be designed as an abstraction of a definitional interpreter [132] using the language’s big-step semantics. These big-step semantics (also referred to as the *natural* semantics), formulated as simple recursive evaluation rules, are often easier to express. Notably, they do not explicitly model the continuation, and instead rely on the underlying continuation of the host language (or formalism) for recursive evaluation rules.

Abstract GC has frequently been integrated into abstract interpreters that employ small-step operational semantics [106, 105]. In contrast, for abstract definitional interpreters that employ big-step definitional semantics, this integration is inherently more challenging. The reason stems from the lack of an explicit continuation component. A direct application of GC requires access to the continuation to ensure that all addresses that are reachable therefrom are not accidentally reclaimed, as they might still be needed later on. That is, addresses reachable from the continuation are traditionally part of the GC root set.

Yet, at the same time, the lack of an explicit continuation on its own brings about key benefits to the precision and scalability of the abstract interpreter. These benefits are widely recognised under the umbrella of *pushdown control flow* models [44, 56]. It should come as no surprise that various attempts [40, 45, 53] have been made to combine the benefits of such models with the benefits of abstract GC, aiming to realise “the best of both worlds”. Unfortunately, these attempts often fall short of that promise, sacrificing some benefits of both abstract GC and pushdown control flow to enable their co-existence. Another issue is that these existing approaches overcomplicate the underlying machinery of the abstract interpreter, requiring significant engineering efforts and rendering formal reasoning about the interpreter’s abstract semantics more challenging.

4.1.1 Goal: The Best of Both Worlds

Both abstract GC and the pushdown control flow of abstract definitional interpreters offer significant precision and performance improvements.

Recall that, for abstract GC, we can enumerate the following advantages:

Avoiding Garbage-Induced Imprecision. An abstract interpreter introduces imprecision by joining different values that are allocated at the same address. Abstract GC can prevent some of those precision-detrimental joins: by freeing up addresses that are unreachable, it prevents future allocations at those addresses from having to join the newly allocated value with the old “garbage values”.

When using abstract counting (Section 2.3.3), abstract GC can also prevent counting dead bindings. That is, by removing garbage, the abstract counts get reset to **0**, and therefore only increased to **1** (instead of ∞) upon the next allocation. Lower abstract counts improve precision by allowing for more strong updates (which in turn may result in more garbage being collected).

Garbage Irrelevance. Another advantage of abstract GC relates not directly to the interpreter’s precision, but to the scalability of the underlying fixpoint algorithm. Specifically, by removing garbage from the heap, the state space explored by the abstract interpreter becomes smaller (i.e., the abstract interpreter only needs to explore the subset of the state space where all garbage is removed). This causes the interpreter to more frequently end up in the same states, allowing the reuse of previously computed analysis results for such states. In contrast, without abstract GC, the interpreter often has to analyse multiple states that are equivalent, but not equal, as they differ only in unreachable (garbage) bindings (i.e., for two states $\hat{\varsigma}_1$ and $\hat{\varsigma}_2$ where $\hat{\varsigma}_1 \neq \hat{\varsigma}_2$ but $\hat{\Gamma}(\hat{\varsigma}_1) = \hat{\Gamma}(\hat{\varsigma}_2)$).

We refer to the examples and results in Chapter 3 to illustrate the impact of these benefits. Only an interpreter that is entirely garbage-free (i.e., eliminates all garbage immediately) enjoys the full benefit of both these advantages.

Similarly, we can enumerate the advantages that are inherent to the push-down control flow of big-step abstract interpreters.

Perfect Stack Precision. The big-step semantics of abstract definitional interpreters no longer model the continuation (i.e., “the stack”), and instead rely on the underlying continuation of the host language (or formalism) for recursive evaluation rules. Since the continuation is no longer modelled in the semantics, an abstraction of these semantics no longer requires an abstraction (i.e., approximation) of the continuation either. Such an abstraction of the continuation would

otherwise cause imprecision in control flow, incorrectly matching callers and returns.¹ In contrast, the pushdown control flow of abstract definitional interpreters entirely avoids this precision loss, resulting in “perfect stack precision” [40].

Context Irrelevance. Germane et al. [53] point out that big-step semantics, such as those of abstract definitional interpreters, also enjoy the benefit of *context irrelevance*. Analogous to garbage irrelevance, context irrelevance ensures that states are not differentiated based on their continuation component (i.e., their “context”). Therefore, when the same evaluation state is encountered in multiple contexts, the analysis results for that state can be reused in every context, improving scalability (i.e., for two states $\hat{\varsigma}_1$ and $\hat{\varsigma}_2$ that are only different because $\hat{\sigma}_{k_{\hat{\varsigma}_1}} \neq \hat{\sigma}_{k_{\hat{\varsigma}_2}}$ or $\hat{a}_{k_{\hat{\varsigma}_1}} \neq \hat{a}_{k_{\hat{\varsigma}_2}}$). The rationale is that just as garbage is not relevant to the evaluation of a current state, in many languages the continuation component also does not contribute to the result of an evaluation. Of course, the latter assumption does not hold when, for instance, first-class continuations are used.

We refer to examples in [44, 45] to illustrate the impact of these benefits. The downside of big-step abstract interpreters is that they cannot directly support first-class continuations (such as those captured by `call/cc`), as they do not explicitly model the continuation component (and instead inherit it implicitly from the defining language). It is known that a CPS transformation of the program under analysis can be used to support such language features [120, 51, 139, 171] by turning continuations into ordinary closures. However, doing so would give up the advantages discussed in this section for the transformed code: context irrelevance would break as evaluation states with different continuations would no longer be equal, perfect stack precision would break when multiple continuation closures are allocated at the same address, and stackless GC (see below) would break as the continuation closure’s references are again included in the GC root set. We describe an alternative strategy to integrate first-class continuations that avoids these limitations in Section 6.3.

In this chapter, we show that the combination of abstract GC and the pushdown control flow of abstract definitional interpreters can not only keep the advantages of both techniques, but can in fact do even better than “the best of both worlds”. That is, we identify an additional benefit that can be exploited by abstract GC when applied to abstract definitional interpreters:

¹In general; there exist variations on AAM, such as AAC [57] and P4F [56] that can recover perfect stack precision by using a sufficiently precise allocator for continuation addresses, albeit at the cost of increased analysis complexity, engineering and/or configurability. Recent work has also shown that m-CFA can exhibit perfect stack precision [51].

Stackless Abstract GC. As exemplified by the work of Germane et al. [53], abstract GC for big-step abstract interpreters can soundly reclaim addresses that are still reachable from the (implicit) continuation for recursive evaluations, restoring these bindings afterwards when the continuation is executed. That is, such “stackless abstract GC” does not take references from the continuation into account, resulting in a smaller GC root set, and therefore also a more aggressive form of abstract GC (and therefore in turn amplifying its aforementioned advantages). We can illustrate this using the following Scheme program as an example:

```
(define (f n)
  (if (even? n) (/ n 2) (let [(r (f (+ n 1)))] (+ r n))))
(f 5)
```

For a context-insensitive abstract interpreter without abstract GC, the recursive call to `f` would allocate the argument `(+ n 1)` at the same address for the parameter `n` as the original argument 5, therefore joining both values at this address and losing precision. However, when abstract GC is applied to a small-step abstract interpreter with explicit continuations (e.g., à la AAM), this precision loss would also not be avoided. The reason is that `n` is still reachable by the continuation (which will bind the result to the variable `r` and then calculate the sum of `r` and `n`) when the recursive call to `f` happens, and therefore its corresponding address cannot be reclaimed. In contrast, when stackless abstract GC is applied to a big-step abstract interpreter, references from the stack do not contribute to the GC root set (i.e., only addresses reachable from the current environment are preserved). As such, `n` is not kept in scope, and its address can be reclaimed before it is reallocated for the recursive call to `f`, avoiding this precision loss.

4.1.2 Approach: Delta Store Semantics

In this work, we present delta store semantics (DSS), a new formalism to integrate abstract GC into abstract definitional interpreters, exploiting all the aforementioned advantages. DSS closely resembles the original big-step abstract semantics, with the only difference being that it returns a *delta store* (containing all changes to the original store) instead of directly returning an updated store as part of the evaluation result. This modification opens up two interesting opportunities:

- It enables a simple and efficient integration of abstract GC into the big-step abstract semantics (similar in spirit to the *effect logs* of Germane et al. [53, 54]). That is, when abstract GC is enabled, the delta store that is computed represents changes w.r.t. a minimal, garbage-collected store. Subsequently, one can *replay* those changes on a non-garbage-collected store, where the bindings that are necessary for the continuation are still present.

- A delta store is a more efficient, minimal representation for the result of an evaluation compared to the full updated store. Just like regular stores, delta stores are joinable. We propose a novel form of store widening for DSS, which still allows for strong updates and (limited) abstract GC. Such store widening involves joining the results of evaluations at the same program point, which is more efficient using delta stores (as we only have to join the changed bindings instead of the entire stores).

These two opportunities form the outline for the core of this chapter. Section 4.3 presents the big-step abstract semantics with delta stores; Section 4.4 and Section 4.5 show how abstract GC and flow-sensitive store widening can be integrated into this semantics, respectively. We evaluate the impact of both in Section 4.6.

4.2 Abstract Definitional Semantics of λ_{ANF}

Similar to how the *Abstracting Abstract Machines (AAM)* approach [105] systematically derives an abstract interpreter from the operational semantics, the *Abstracting Definitional Interpreters (ADI)* approach [40] systematically derives abstract interpreters starting from the definitional semantics. We follow the ADI approach and formulate the abstract big-step (definitional) semantics of λ_{ANF} , equivalent to the abstract small-step semantics of λ_{ANF} presented in Section 2.3.

We also follow the approach originally proposed by Darais et al. [40] to integrate abstract GC into these semantics, and in doing so point out why the proposed approach falls short of the aforementioned “best of both worlds” objectives.

Evaluation Rules. For an abstract definitional interpreter, the semantics are formulated as recursive evaluation rules using a big-step relation $(\Downarrow) \subseteq \widehat{\text{Config}} \times \widehat{\text{Result}}$. We define input configurations to evaluate $(\widehat{\text{Config}})$ and the result of the evaluation for that configuration $(\widehat{\text{Result}})$ as follows:

$$\widehat{\text{conf}} \in \widehat{\text{Config}} = \widehat{\text{Exp}} \times \widehat{\text{Env}} \times \widehat{\text{Store}} \times \widehat{\text{Time}} \quad \widehat{\text{res}} \in \widehat{\text{Result}} = \widehat{\text{Val}} \times \widehat{\text{Store}}$$

A configuration $\langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle$ contains an expression e to evaluate using the environment $\hat{\rho}$, store $\hat{\sigma}$ and the current timestamp $\hat{t}$. In contrast to states in $\hat{\Sigma}$, configurations crucially do not carry a continuation component (i.e., a continuation store $\hat{\sigma}_k$ or address $\hat{a}_k$), which is key to achieve context irrelevance (as configurations with the same expression, environment, store, and timestamp but used in different evaluation contexts are still considered equal) and avoids a loss in stack precision (since there is no need to approximate this component). The result of an evaluation $\langle \hat{v}, \hat{\sigma}' \rangle$ holds the resulting value $\hat{v}$ along with the updated store $\hat{\sigma}'$.

We define the big-step evaluation relation ($\Downarrow$) as the minimal relation closed over the evaluation rules shown below. Instead of working with explicit continuations, the evaluation rules now employ recursion. As with the small-step transition relation ($\Rightarrow$), there is nondeterminism due to the over-approximating behaviour.

$$\begin{array}{c}
 \frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma} \rangle} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\sigma}' = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{set } x := ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle} \quad [\text{E-SET}] \\
 \\
 \frac{\begin{array}{c} (\hat{f}, -) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad (\lambda x.e, \hat{\rho}') \in \hat{f} \\ \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}') \\ \langle e, \hat{\rho}'[x \mapsto \hat{a}_x], \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_x, \hat{v}_x), \hat{t}' \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle \end{array}}{\langle f(ae), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{c} (-, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (-, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\ \hat{v} = (\emptyset, \widehat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \end{array}}{\langle op(ae_1, ae_2), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma} \rangle} \quad [\text{E-BINOP}] \\
 \\
 \frac{\begin{array}{c} \langle e_1, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}_x, \hat{\sigma}' \rangle \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \\ \langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x), \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}'' \rangle \end{array}}{\langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}'' \rangle} \quad [\text{E-LET}]
 \end{array}$$

Note that strong updates are already supported in these semantics, since we are reusing the abstract-counting definitions of $\widehat{\text{extend}}$ and $\widehat{\text{update}}$ from Section 2.3.3.

Abstract Garbage Collection. For the small-step abstract interpreter, we computed the GC root set for the current state $\hat{\varsigma}$ using the addresses directly reachable from the control component ($\widehat{\mathcal{T}}_{\text{Control}}(\hat{e}_{\hat{\varsigma}})$) and the current root of the continuation component ($\hat{a}_{k_{\hat{\varsigma}}}$) (Section 2.3.2). For the abstract definitional interpreter presented here, we do not have access to the latter (or the continuation store $\hat{\sigma}_k$), as the continuation component is not explicitly reified. Nevertheless, addresses reachable from the continuation cannot just be ignored: if they are not included in the GC root set, addresses that are used later on may accidentally be collected. The updated store $\hat{\sigma}'$ of an evaluation step $\langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle$ would then no longer be

safe to continue evaluation with. For instance, in the E-LET rule, it is crucial that the updated store $\hat{\sigma}'$ (returned after evaluating e_1) preserves all original bindings reachable from $\hat{\rho}$, so that it can safely be used to evaluate e_2 .

In order to address this, Darais et al. [40] show how abstract GC can still be integrated into an abstract definitional interpreter by explicitly passing along all addresses reachable from the implicit continuation (i.e., all addresses that need to be preserved, as they may be used after the current evaluation step). Concretely, this requires passing along a set of addresses $\hat{\psi}$ in each configuration:

$$\widehat{conf} \in \widehat{\text{Config}} = \widehat{Exp} \times \widehat{\text{Env}} \times \widehat{\text{Store}} \times \widehat{\text{Time}} \times \mathcal{P}(\widehat{\text{Addr}})$$

Incorporating GC into the evaluation rules then requires two modifications:

- Each recursive evaluation step passes along the set of stack addresses $\hat{\psi}$ that need to be preserved, along with additional addresses that are still needed to continue with the remaining evaluation steps in that rule. The latter corresponds to the addresses that would previously be directly reachable from the continuation (i.e., using $\widehat{\mathcal{T}}_{\text{Kont}}$) allocated in the corresponding rule of the small-step semantics. Due to our simplification to ANF, the only rule with multiple recursive evaluation steps is the E-LET rule; addresses still required for the second evaluation step (in this case, $\widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho})$) are therefore added to $\hat{\psi}$ for the first evaluation step. As the second evaluation step is also the final one (i.e., e_2 is evaluated in *tail position*, so it does not grow the implicit continuation), no additional addresses need to be preserved.
- We add a new rule to collect garbage: for a configuration $\langle e, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle$, it uses $\hat{\psi} \cup \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho})$ as the root set to GC the store that expression e is evaluated with, and subsequently also cleans up the updated store $\hat{\sigma}'$ of the result $\langle \hat{v}, \hat{\sigma}' \rangle$ using the GC root set $\hat{\psi} \cup \widehat{\mathcal{T}}_{\text{Val}}(\hat{v})$.² To interleave this GC rule with the existing evaluation rules, we introduce a new relation ($\Downarrow_{\hat{\Gamma}}$):

$$\frac{\begin{array}{l} \hat{\sigma}_{\text{gc}} = \hat{\sigma} |_{\widehat{\mathcal{R}}_{(\hat{\psi} \cup \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}))}} \\ \langle e, \hat{\rho}, \hat{\sigma}_{\text{gc}}, \hat{t}, \hat{\psi} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle \\ \hat{\sigma}'_{\text{gc}} = \hat{\sigma}' |_{\widehat{\mathcal{R}}_{(\hat{\psi} \cup \widehat{\mathcal{T}}_{\text{Val}}(\hat{v}))}} \end{array}}{\langle e, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle \Downarrow_{\hat{\Gamma}} \langle \hat{v}, \hat{\sigma}'_{\text{gc}} \rangle} \quad [\text{E-GC}]$$

²Note that we are able to collect more garbage here compared to the original work of Darais et al. [40], as we GC not only the result store, but also the configuration store.

Subsequently, we modify the other evaluation rules to use $(\Downarrow_{\hat{\Gamma}})$ instead of $(\Downarrow)$ for recursive evaluation steps. This effectively results in the big-step equivalent of the “GC at every step” policy used in $(\rightrightarrows_{\hat{\Gamma}})$ (Section 2.3.2).

The updated evaluation rules are shown below:

$$\begin{array}{c}
 \frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle ae, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle \Downarrow \langle \hat{v}, \hat{\sigma} \rangle} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\sigma}' = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{set } x := ae, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle} \quad [\text{E-SET}] \\
 \\
 \frac{\begin{array}{c} (\hat{f}, -) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad (\lambda x.e, \hat{\rho}') \in \hat{f} \\ \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}') \\ \langle e, \hat{\rho}'[x \mapsto \hat{a}_x], \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_x, \hat{v}_x), \hat{t}', \hat{\psi} \rangle \Downarrow_{\hat{\Gamma}} \langle \hat{v}, \hat{\sigma}' \rangle \end{array}}{\langle f(ae), \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{c} (-, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (-, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\ \hat{v} = (\emptyset, \hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \end{array}}{\langle op(ae_1, ae_2), \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle \Downarrow \langle \hat{v}, \hat{\sigma} \rangle} \quad [\text{E-BINOP}] \\
 \\
 \frac{\begin{array}{c} \langle e_1, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \cup \widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}) \rangle \Downarrow_{\hat{\Gamma}} \langle \hat{v}_x, \hat{\sigma}' \rangle \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \\ \langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x), \hat{t}, \hat{\psi} \rangle \Downarrow_{\hat{\Gamma}} \langle \hat{v}, \hat{\sigma}'' \rangle \end{array}}{\langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{\psi} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}'' \rangle} \quad [\text{E-LET}]
 \end{array}$$

While this approach can safely collect garbage to improve precision, it still loses out on the full advantages of both abstract GC and pushdown control flow:

- First, the abstract interpreter no longer enjoys context irrelevance. While the continuation itself is not directly part of a configuration, the set of addresses that are reachable from the continuation are. Configurations that only differ in the context in which they are used (i.e., they have different continuations), and that would otherwise be equal, may therefore be analysed multiple times by the analysis (i.e., when the set of reachable addresses $\hat{\psi}$ is also different, leading to multiple configurations).

- Second, the abstract interpreter cannot take advantage of the pushdown control flow to collect more garbage (i.e., it does not allow for stackless GC). The reason is that it cannot reclaim any bindings reachable from the continuation (as these are kept alive by $\hat{\psi}$), even though these are not actually used for the current evaluation step. An optimal approach would be able to more aggressively collect garbage using a smaller root set (i.e., without $\hat{\psi}$), only adding back the bindings reachable from $\hat{\psi}$ *after* the evaluation step to construct the updated store to continue evaluation with.

Program Semantics. As with the small-step semantics, we can define a function $\widehat{\text{eval}} : \text{Exp} \rightarrow \mathcal{P}(\widehat{\text{Val}})$ that returns all possible return values:

$$\widehat{\text{eval}}(e) = \{\hat{v} \mid \exists \hat{\sigma} : \langle e, [], \perp_{\widehat{\text{Store}}}, \hat{t}_0, \emptyset \rangle \Downarrow_{\hat{\Gamma}} \langle \hat{v}, \hat{\sigma} \rangle\}$$

For now, we only define the program semantics declaratively, and leave open the algorithm used to actually compute them. We refer to the work of Darais et al. [40] for a (simple, but inefficient) cache-based fixpoint algorithm to compute the (finite) big-step relation ($\Downarrow_{\hat{\Gamma}}$). The same paper also shows how the evaluation rules can be instrumented to compute the collecting semantics. In Chapter 5, we propose a more efficient worklist algorithm to compute the least fixpoint.

4.3 Delta Store Semantics

We now present the core foundation of delta store semantics (DSS). DSS closely resembles the standard big-step semantics presented in Section 4.2. The main difference is that evaluation now returns a *delta store* (representing changes to the original store) instead of the entire updated store.

We first introduce delta stores in Section 4.3.1, along with a set of operations on delta stores defining how they are *joined* and *composed*. We also show how a delta store can be applied to the original store, in order to reconstruct the updated store. In Section 4.3.2, we then modify the abstract big-step semantics for λ_{ANF} , resulting in an equivalent semantics formulated using delta stores.

4.3.1 Delta Stores

A delta store captures bindings that were changed with respect to a given store $\hat{\sigma}$. Put differently, it is the subset of the updated store containing all bindings that $\hat{\sigma}$ has potentially³ been extended or updated with.

Formally, we define delta stores similarly to regular stores:

$$\hat{\delta} \in \widehat{\text{Delta}} = \widehat{\text{Addr}} \rightarrow (\widehat{\text{Val}} \times \widehat{\text{Count}})$$

The difference is that a delta store is represented as a *partial* function, since it only maps addresses that have been modified w.r.t. the original store.

In what follows, we show how delta stores can be applied, composed, and joined.

Application. A delta store $\hat{\delta}$, representing changes over an original store $\hat{\sigma}$, can be applied to $\hat{\sigma}$ in order to construct the updated store in which these changes are directly integrated. Since both stores and delta stores are represented as functions, we can use the mathematical *override* operator ($\triangleright$) for this purpose:

$$(f_1 \triangleright f_2)(a) = \begin{cases} f_1(a) & \text{if } a \in \text{dom}(f_1) \\ f_2(a) & \text{if } a \notin \text{dom}(f_1) \wedge a \in \text{dom}(f_2) \end{cases}$$

That is, we can write $\hat{\delta} \triangleright \hat{\sigma}$ to apply the changes of delta store $\hat{\delta}$ to the original store $\hat{\sigma}$. Note that the result of this operation is a $\widehat{\text{Store}}$, as $\text{dom}(\hat{\sigma}) = \widehat{\text{Addr}}$, so that $\hat{\delta} \triangleright \hat{\sigma}$ is always a total function $\widehat{\text{Addr}} \rightarrow (\widehat{\text{Val}} \times \widehat{\text{Count}}) = \widehat{\text{Store}}$. A lookup in this store will first look for the address in the delta store $\hat{\delta}$, and for unchanged bindings (i.e., that are not in $\hat{\delta}$) fall back to the original store $\hat{\sigma}$.

Composition. We also require an operator to combine multiple delta stores that represent a sequence of changes (e.g., resulting from the evaluation of multiple expressions in a sequence). That is, given a delta store $\hat{\delta}_1$ (representing changes w.r.t. a store $\hat{\sigma}$) followed by a delta store $\hat{\delta}_2$ (representing changes to the result of $\hat{\delta}_1 \triangleright \hat{\sigma}$), we want to be able to combine these changes into a single delta store.

To this end, we can use the same override operator ($\triangleright$) as an operator for composition. A series of changes represented in order by delta stores $\hat{\delta}_1, \hat{\delta}_2, \dots, \hat{\delta}_n$ can then be composed into a single delta store $\hat{\delta}_n \triangleright \dots \triangleright \hat{\delta}_2 \triangleright \hat{\delta}_1$, which (when applied

³A delta store may also contain updated bindings that happen to have the same value as in the original store $\hat{\sigma}$, although this makes its usage slightly less efficient.

to a store $\hat{\sigma}$) will integrate all changes of these delta stores in that same order. Note that, when applied to two delta stores $\hat{\delta}_1$ and $\hat{\delta}_2$, the result of the composition is again a delta store (i.e., a partial function) with $\text{dom}(\hat{\delta}_2 \triangleright \hat{\delta}_1) = \text{dom}(\hat{\delta}_1) \cup \text{dom}(\hat{\delta}_2)$.

Join. Delta stores are joinable, but only when they represent changes over the same store $\hat{\sigma}$, so that the resulting delta store also represents changes over $\hat{\sigma}$. Therefore, we define a join operator $(\sqcup_{\hat{\sigma}})$ parameterised over some store $\hat{\sigma}$:

$$(\hat{\delta}_1 \sqcup_{\hat{\sigma}} \hat{\delta}_2)(\hat{a}) = \begin{cases} \hat{\delta}_1(\hat{a}) \sqcup \hat{\delta}_2(\hat{a}) & \text{if } \hat{a} \in \text{dom}(\hat{\delta}_1) \wedge \hat{a} \in \text{dom}(\hat{\delta}_2) \\ \hat{\delta}_1(\hat{a}) \sqcup \hat{\sigma}(\hat{a}) & \text{if } \hat{a} \in \text{dom}(\hat{\delta}_1) \wedge \hat{a} \notin \text{dom}(\hat{\delta}_2) \\ \hat{\sigma}(\hat{a}) \sqcup \hat{\delta}_2(\hat{a}) & \text{if } \hat{a} \notin \text{dom}(\hat{\delta}_1) \wedge \hat{a} \in \text{dom}(\hat{\delta}_2) \end{cases}$$

Note that $\text{dom}(\hat{\delta}_1 \sqcup_{\hat{\sigma}} \hat{\delta}_2) = \text{dom}(\hat{\delta}_1) \cup \text{dom}(\hat{\delta}_2)$. When joining delta stores, it is important to consider what happens when an address is in one delta store, but not the other. In this case, we join the binding of the delta store with the binding of the original store $\hat{\sigma}$, since the lack of a binding in the other delta store implies no change w.r.t. the original store (hence, the original binding is preserved).

The join operator $(\sqcup_{\hat{\sigma}})$ is useful to handle non-determinism in the abstract semantics, as the results of multiple program paths (which each include a delta store) following a non-deterministic choice can be joined together into a single result. We will make use of this operator later in Section 4.5 when we integrate flow-sensitive store widening into the abstract semantics using delta stores.

4.3.2 Big-Step Abstract Semantics with Delta Stores for λ_{ANF}

We now formulate the big-step abstract semantics for λ_{ANF} using delta stores. The main difference to the big-step semantics presented in Section 4.2 is that all modifications to the store are now expressed as delta stores. We first adapt the auxiliary functions that modify the store, extend and update, to return a delta store (with a single change) instead of an updated store.

$$\begin{aligned} \widehat{\text{extend}}(\hat{\sigma}, \hat{a}, \hat{v}) &= [\hat{a} \mapsto \langle \hat{v}_a \sqcup \hat{v}, \widehat{\text{inc}}(\hat{q}_a) \rangle] \\ \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v}) &= \begin{cases} [\hat{a} \mapsto \langle \hat{v}, \mathbf{1} \rangle] & \text{if } \hat{q}_a = \mathbf{1} \\ [\hat{a} \mapsto \langle \hat{v}_a \sqcup \hat{v}, \hat{q}_a \rangle] & \text{otherwise} \end{cases} \\ \text{where } \langle \hat{v}_a, \hat{q}_a \rangle &= \hat{\sigma}(\hat{a}) \end{aligned}$$

Note that these definitions have remained mostly unchanged. The only difference is that the change is now expressed as a “portable” delta store, as opposed to being directly integrated into the store that is extended or updated.

Next, we introduce a new big-step evaluation relation ($\Downarrow^\Delta$), which returns a delta store instead of an updated store as part of the result. That is, we define:

$$\widehat{\text{res}} \in \widehat{\text{Result}} = \widehat{\text{Val}} \times \widehat{\text{Delta}}$$

The updated evaluation rules for ($\Downarrow^\Delta$) are given below:

$$\begin{array}{c}
 \frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, [] \rangle} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\delta} = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{set } x := ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, \hat{\delta} \rangle} \quad [\text{E-SET}] \\
 \\
 \frac{\begin{array}{l} (\hat{f}, _) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad (\lambda x.e, \hat{\rho}') \in \hat{f} \\ \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}') \quad \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_x, \hat{v}_x) \\ \langle e, \hat{\rho}'[x \mapsto \hat{a}_x], \hat{\delta}_x \triangleright \hat{\sigma}, \hat{t}' \rangle \Downarrow^\Delta \langle \hat{v}, \hat{\delta} \rangle \end{array}}{\langle f(ae), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, \hat{\delta} \triangleright \hat{\delta}_x \rangle} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{l} (_, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (_, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\ \hat{v} = (\emptyset, \hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \end{array}}{\langle op(ae_1, ae_2), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, [] \rangle} \quad [\text{E-BINOP}] \\
 \\
 \frac{\begin{array}{l} \langle e_1, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}_x, \hat{\delta}_1 \rangle \quad \hat{\sigma}' = \hat{\delta}_1 \triangleright \hat{\sigma} \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \\ \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x) \quad \langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \hat{\delta}_x \triangleright \hat{\sigma}', \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, \hat{\delta}_2 \rangle \end{array}}{\langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, \hat{\delta}_2 \triangleright \hat{\delta}_x \triangleright \hat{\delta}_1 \rangle} \quad [\text{E-LET}]
 \end{array}$$

When evaluating an atomic expression (E-ATOM) or numerical operation (E-BINOP), the store is not modified, and hence the empty delta store $[]$ is returned.

When evaluating a variable assignment (E-SET), a delta store is returned representing the update to that variable binding. When evaluating an application (E-APP), the store is first extended with a binding for the argument (represented by $\hat{\delta}_x$), and afterwards also potentially modified during the evaluation of the function body (represented by $\hat{\delta}$). The resulting delta store is therefore the composed delta store $\hat{\delta} \triangleright \hat{\delta}_x$. Similarly, when evaluating a let expression (E-LET), the resulting delta store composes the changes made by the evaluation of the right-hand side expression e_1 , the binding of the variable x , and the evaluation of the body e_2 .

One can show that the abstract big-step semantics defined with $(\Downarrow^\Delta)$ is equivalent to the semantics defined with $(\Downarrow)$, as stated by Theorem 10.

Theorem 10 $\forall e, \hat{\rho}, \hat{\sigma}, \hat{\sigma}', \hat{v}, \hat{t} :$

$$\langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow \langle \hat{v}, \hat{\sigma}' \rangle \iff \exists \hat{\delta} : \langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow^\Delta \langle \hat{v}, \hat{\delta} \rangle \wedge \hat{\sigma}' = \hat{\delta} \triangleright \hat{\sigma}$$

Proof. A proof for Theorem 10 can be constructed by induction on $(\Downarrow)$ (for the $(\implies)$ direction) and on $(\Downarrow^\Delta)$ (for the $(\impliedby)$ direction). $\square$

Note that Theorem 10 shows equivalence between the semantics of ADI and DSS. We can thus forgo constructing dedicated soundness and termination proofs for DSS, as these have already been established for ADI in existing work [40, 83].

4.4 Integrating Abstract GC

We now make use of these new big-step semantics using delta stores to integrate abstract GC. By using delta stores (instead of returning the updated store), it is now safe to evaluate an expression using an input store that has been garbage collected without taking references from the implicit continuation into account (i.e., they can be GC'd). The key insight is that the computed delta store can later be *replayed* on the original store that has not yet been garbage collected, and in which the bindings necessary for the continuation are still present.

To enable this process, we first introduce such a replay operation for delta stores in Section 4.4.1. Using this new operation, we then integrate abstract GC into the delta store semantics in Section 4.4.2. In Section 4.4.3, we show that, for programs that do not make use of variable assignments, the resulting form of abstract *stackless* GC is not just equivalent, but in fact superior to the traditional form of abstract GC found in small-step abstract interpreters.

4.4.1 Replaying Delta Stores

A key ingredient to allow the integration of abstract GC into the big-step semantics is the *replay* operation. It transforms a delta store $\hat{\delta}$ (which was computed

using a store where bindings for the continuation may have been collected) into a delta store $\hat{\delta}'$ that “restores” the bindings necessary for the continuation (found in the original store $\hat{\sigma}$). That is, it simulates (“replays”) the changes captured by $\hat{\delta}$ on $\hat{\sigma}$, computing the changes that would have occurred with respect to $\hat{\sigma}$.

To illustrate the purpose of the replay operation, consider an abstract interpretation (with abstract GC) for the following Scheme program:

```
(define (make-adder n)
  (lambda (x) (+ x n)))
(let ((f1 (make-adder 1)))
  (let ((f2 (make-adder 2)))
    (f2 (f1 0)))))
```

Assuming a context-insensitive allocation policy, the abstract interpreter will allocate the same address for the variable **n** (henceforth denoted as $\textcircled{n}$) in both calls to **make-adder**. The first call will return a delta store containing the binding $\textcircled{n} \rightarrow 1$. This binding can (temporarily) be garbage collected when evaluating the second call: while it is still necessary for the continuation of that second call, its evaluation itself does not require this binding. The second call will therefore return a delta store containing the binding $\textcircled{n} \rightarrow 2$. However, after returning from the second call, we cannot just continue evaluation using a store that binds $\textcircled{n}$ to 2. That is, we first need to use the replay operation to restore the old binding as well, resulting in a store where $\textcircled{n}$ is bound to the join of 1 and 2.

Formally, we define $\widehat{\text{replay}} : \widehat{\text{Delta}} \times \widehat{\text{Store}} \times \mathcal{P}(\widehat{\text{Addr}}) \rightarrow \widehat{\text{Delta}}$ as follows:

$$\widehat{\text{replay}}(\hat{\delta}, \hat{\sigma}, \hat{A})(\hat{a}) = \begin{cases} \hat{\delta}(\hat{a}) & \text{if } \hat{a} \in \text{dom}(\hat{\delta}) \wedge \hat{a} \notin \hat{A} \\ \langle \hat{v}_{\hat{\sigma}} \sqcup \hat{v}_{\hat{\delta}}, \widehat{\text{inc}}(\hat{q}_{\hat{\sigma}}) \rangle & \text{if } \hat{a} \in \text{dom}(\hat{\delta}) \wedge \hat{a} \in \hat{A} \wedge \hat{q}_{\hat{\delta}} = \mathbf{1} \\ \langle \hat{v}_{\hat{\sigma}} \sqcup \hat{v}_{\hat{\delta}}, \infty \rangle & \text{if } \hat{a} \in \text{dom}(\hat{\delta}) \wedge \hat{a} \in \hat{A} \wedge \hat{q}_{\hat{\delta}} = \infty \end{cases}$$

where $\langle \hat{v}_{\hat{\sigma}}, \hat{q}_{\hat{\sigma}} \rangle = \hat{\sigma}(\hat{a})$ **and** $\langle \hat{v}_{\hat{\delta}}, \hat{q}_{\hat{\delta}} \rangle = \hat{\delta}(\hat{a})$

Note that replaying a delta store preserves the domain it is defined over (i.e., $\forall \hat{\delta}, \hat{\sigma}, \hat{A} : \text{dom}(\widehat{\text{replay}}(\hat{\delta}, \hat{\sigma}, \hat{A})) = \text{dom}(\hat{\delta})$).

Function $\widehat{\text{replay}}$ takes three arguments: the delta store $\hat{\delta}$ to be replayed, the original store $\hat{\sigma}$ (on which $\hat{\delta}$ is replayed), and a set $\hat{A}$ of all addresses in $\hat{\delta}$ that were *allocated* (i.e., representing (abstractions of) new addresses) during the computation of $\hat{\delta}$. The case where an address $\hat{a}$ is not in $\hat{A}$, but is present in $\hat{\delta}$ implies that

the change represents an update to an existing binding (since no new binding for $\hat{a}$ was allocated). That is, address $\hat{a}$ was not collected, and therefore $\widehat{\text{replay}}$ does not need to restore the original binding of $\hat{a}$ in $\hat{\sigma}$ (as we know that this binding was still present when the update happened). In fact, preserving $\hat{\delta}(\hat{a})$ *as is* is important to maintain strong updates. In case address $\hat{a}$ in $\hat{\delta}$ may come from a new allocation (i.e., $\hat{a} \in \hat{A}$), we join the old value in $\hat{\sigma}(\hat{a})$ with the updated one in $\hat{\delta}(\hat{a})$ (as $\widehat{\text{extend}}$ would have done for a new allocation on $\hat{\sigma}$). When $\hat{q}_{\hat{\delta}} = \mathbf{1}$, we know that $\hat{a}$ was only allocated once, and hence we increase the abstract count in $\hat{\sigma}$ by 1 (using $\widehat{\text{inc}}$). When $\hat{q}_{\hat{\delta}} = \infty$, we know that $\hat{a}$ may have been allocated multiple times, so the abstract count becomes ∞ .

4.4.2 Delta Store Semantics with Abstract GC

We now modify the big-step evaluation relation $(\Downarrow^{\Delta})$ to integrate abstract GC. In order to do so, we need to keep track of which addresses in $\hat{\delta}$ may have been *allocated* and which may have been *updated*. Keeping track of a set of allocated addresses $\hat{A}$ is necessary to support the replay operation (see above). Keeping track of a set of updated addresses $\hat{U}$ is necessary to correctly collect garbage in the resulting delta store, which needs to maintain all bindings reachable from the resulting value *and* all existing bindings that were updated during the evaluation (in order to ensure that side effects that happened during evaluation are not lost). We add these two sets of addresses directly to the result of the evaluation:

$$\widehat{\text{res}} \in \widehat{\text{Result}} = \widehat{\text{Val}} \times \widehat{\text{Delta}} \times \mathcal{P}(\widehat{\text{Addr}}) \times \mathcal{P}(\widehat{\text{Addr}})$$

Next, we adapt the evaluation rules to integrate abstract GC as follows:

- For each evaluation rule, we trivially construct both $\hat{A}$ and $\hat{U}$ as the set of addresses allocated (resp. updated) directly in that rule, combined with those allocated (resp. updated) in recursive evaluation steps.
- In the E-LET rule (the only evaluation rule for λ_{ANF} with a non-tail recursive evaluation step), we ensure that the delta store computed by the first recursive evaluation step is replayed, so that the bindings needed for the second recursive evaluation step are restored. All other recursive evaluation steps in the evaluation rules do not have an additional (implicit) continuation for which bindings are to be preserved, and therefore do not require their delta store to be replayed (doing so would unnecessarily add imprecision).
- Analogous to the integration of abstract GC into $(\Downarrow)$ (see Section 4.2), we add an evaluation relation $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ with the following E-GC rule.

$$\begin{array}{c}
\hat{\sigma}_{\text{gc}} = \hat{\sigma} |_{\widehat{\mathcal{R}}_{(\hat{\sigma})}}(\widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho})) \\
\langle e, \hat{\rho}, \hat{\sigma}_{\text{gc}}, \hat{t} \rangle \Downarrow_{\hat{\Delta}}^{\Delta} \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle \\
\hat{\delta}_{\text{gc}} = \hat{\delta} |_{\widehat{\mathcal{R}}_{(\hat{\sigma}_{\text{gc}})}}(\widehat{\mathcal{T}}_{\text{Val}}(\hat{v}) \cup \hat{U}_{\text{gc}}) \\
\frac{\hat{A}_{\text{gc}} = \hat{A} \cap \text{dom}(\hat{\delta}_{\text{gc}}) \quad \hat{U}_{\text{gc}} = \{\hat{a} \in \hat{U} \mid \hat{\sigma}_{\text{gc}}(\hat{a}) \neq \perp\}}{\langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_{\text{gc}}, \hat{A}_{\text{gc}}, \hat{U}_{\text{gc}} \rangle} \quad [\text{E-GC}]
\end{array}$$

That is, rule E-GC collects garbage in both the input store $\hat{\sigma}$ as well as the output delta store $\hat{\delta}$ (computing reachable addresses over $\hat{\delta} \triangleright \hat{\sigma}_{\text{gc}}$, as bindings in the delta store $\hat{\delta}$ may reference those in $\hat{\sigma}_{\text{gc}}$ and vice versa). It also restricts the sets of addresses $\hat{A}$ and $\hat{U}$: $\hat{A}$ can be restricted by only keeping addresses that are actually in $\hat{\delta}_{\text{gc}}$ (removing garbage-collected allocations), and $\hat{U}$ can be restricted by only keeping addresses that have an existing binding in the store $\hat{\sigma}_{\text{gc}}$. The crucial difference with the E-GC rule of $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ is that the stores are garbage collected using only the references from the environment (for $\hat{\sigma}$) and the returned and updated values as the GC root set (for $\hat{\delta}$). References from the continuation, previously kept alive using $\hat{\psi}$, now no longer need to be taken into account, leading to more garbage being collected and therefore amplifying the benefits of abstract GC. We subsequently replace $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ with $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ for all recursive evaluation steps, so that GC is applied at every step.

The resulting evaluation rules for $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ are given below:

$$\begin{array}{c}
\frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, [], \emptyset, \emptyset \rangle} \quad [\text{E-ATOM}] \\
\\
\frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\delta} = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{set } x := ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}, \emptyset, \{\hat{a}\} \rangle} \quad [\text{E-SET}] \\
\\
\frac{\begin{array}{c} (\hat{f}, _) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad (\lambda x.e, \hat{\rho}') \in \hat{f} \\ \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}') \quad \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_x, \hat{v}_x) \end{array}}{\langle e, \hat{\rho}'[x \mapsto \hat{a}_x], \hat{\delta}_x \triangleright \hat{\sigma}, \hat{t}' \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle} \quad [\text{E-APP}] \\
\frac{}{\langle f(ae), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta} \triangleright \hat{\delta}_x, \{\hat{a}_x\} \cup \hat{A}, \hat{U} \rangle}
\end{array}$$

$$\begin{array}{c}
 \dfrac{
 \begin{array}{c}
 (-, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (-, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\
 \hat{v} = (\emptyset, \hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2))
 \end{array}
 }{
 \langle op(ae_1, ae_2), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, [], \emptyset, \emptyset \rangle
 } \quad [\text{E-BINOP}]
 \\
 \\
 \dfrac{
 \begin{array}{c}
 \langle e_1, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}_x, \hat{\delta}_1, \hat{A}_1, \hat{U}_1 \rangle \quad \hat{\delta}'_1 = \widehat{\text{replay}}(\hat{\delta}_1, \hat{\sigma}, \hat{A}_1) \\
 \hat{\sigma}' = \hat{\delta}'_1 \triangleright \hat{\sigma} \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \quad \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x) \\
 \langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \hat{\delta}_x \triangleright \hat{\sigma}', \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_2, \hat{A}_2, \hat{U}_2 \rangle \\
 \hat{A} = \hat{A}_1 \cup \{\hat{a}_x\} \cup \hat{A}_2 \quad \hat{U} = \hat{U}_1 \cup \hat{U}_2
 \end{array}
 }{
 \langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_2 \triangleright \hat{\delta}_x \triangleright \hat{\delta}'_1, \hat{A}, \hat{U} \rangle
 } \quad [\text{E-LET}]
 \end{array}$$

4.4.3 Comparison to Traditional Abstract GC

It is interesting to compare the impact of our formulation of abstract GC for DSS with the impact of existing “traditional” abstract GC, as pioneered by Might et al. for small-step abstract interpreters [106]. Intuitively, we have already established that abstract GC in the DSS semantics (i.e., using $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ of Section 4.4.2) can be more efficient than abstract GC integrated into the big-step semantics without DSS (i.e., using $(\hat{\Downarrow}_{\hat{\Gamma}})$ of Section 4.2): being able to omit the continuation references $\hat{\psi}$ results in a smaller GC root set and more garbage being collected, therefore amplifying the beneficial effects of abstract GC.

We now show that the same reasoning also holds for a comparison with abstract GC in the small-step semantics (as presented in Section 2.3.2) for pure (ANF-style) λ -calculus (i.e., a subset of λ_{ANF} without variable assignment, which we refer to as λ_{ANF}^-). That is, because *stackless* GC can collect more garbage, the result obtained through a big-step evaluation using $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ is guaranteed to be at least as precise as the corresponding result obtained by a small-step interpreter using $(\hat{\Rightarrow}_{\hat{\Gamma}})$ for every program in λ_{ANF}^- . Theorem 11 states this claim more formally⁴:

Theorem 11 $\forall e \in \lambda_{\text{ANF}}^-, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{v}, \hat{\delta}, \hat{A}, \hat{U}, \hat{\sigma}_k, \hat{a}_k : \langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \hat{\Downarrow}_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle$
 $\rightarrow \exists \hat{v}', \hat{\sigma}', \hat{\sigma}'_k : \langle \text{ev}(e, \hat{\rho}, \hat{t}), \hat{\sigma}, \hat{\sigma}_k, \hat{a}_k \rangle \hat{\Rightarrow}_{\hat{\Gamma}}^* \langle \text{val}(\hat{v}'), \hat{\sigma}', \hat{\sigma}'_k, \hat{a}_k \rangle \wedge \hat{v} \sqsubseteq \hat{v}'$

That is, when an expression e evaluates to $\hat{v}$ (using an environment $\hat{\rho}$, store $\hat{\sigma}$ and timestamp $\hat{t}$), there exists at least one sequence of evaluation steps for the small-step abstract interpreter that evaluates the same expression e to a less precise

⁴We slightly abuse notation and assume that both the small-step and big-step semantics have been instantiated for λ_{ANF}^- . That is, in what follows, all expressions in the interpreters’ components (including those in closures and continuations) are restricted to λ_{ANF}^- .

(or equally precise) value $\hat{v}'$. We can match the small-step state $\hat{\zeta}$ evaluating e with the state $\hat{\zeta}'$ holding its result $\hat{v}'$ by ensuring that $\hat{a}_{k_{\hat{\zeta}}} = \hat{a}_{k_{\hat{\zeta}'}}$ (i.e., when the abstract interpreter reaches the same continuation $\hat{a}_k$ again with a return value).

We can prove Theorem 11 using mutual induction on $(\Downarrow^\Delta)$ and $(\Downarrow_F^\Delta)$. Doing so, however, first requires generalising this theorem in order to strengthen the induction hypothesis, resulting in Theorem 12.

Theorem 12 $\forall e \in \lambda_{ANF}^-, \hat{\rho}, \hat{\sigma}, \hat{t}, \hat{v}, \hat{\delta}, \hat{A}, \hat{U}, \hat{\zeta} = \langle ev(e, \hat{\rho}, \hat{t}), \hat{\sigma}_{\hat{\zeta}}, \hat{\sigma}_{k_{\hat{\zeta}}}, \hat{a}_k \rangle,$
 $\hat{\mathcal{R}}_{\hat{\rho}} = \hat{\mathcal{R}}_{(\hat{\sim}_{\hat{\sigma}})}(\hat{\mathcal{T}}_{Env}(\hat{\rho})), \hat{\sigma}_{gc} = \hat{\sigma}|_{\hat{\mathcal{R}}_{\hat{\rho}}}, \hat{\mathcal{R}}_{\hat{v}} = \hat{\mathcal{R}}_{(\hat{\sim}_{\hat{\delta} \triangleright \hat{\sigma}_{gc}})}(\hat{\mathcal{T}}_{Val}(\hat{v})) :$
 $\langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle \Downarrow_F^\Delta \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle \wedge (\hat{\sigma}_{gc} \sqsubseteq \hat{\sigma}_{\hat{\zeta}}) \wedge (\hat{\Gamma}(\hat{\zeta}) = \hat{\zeta}) \rightarrow \exists \hat{\zeta}' = \langle \mathbf{val}(\hat{v}'), \hat{\sigma}_{\hat{\zeta}'}, \hat{\sigma}_{k_{\hat{\zeta}'}} , \hat{a}_k \rangle :$
 $\hat{\zeta} \xrightarrow{\ast}_{\hat{\Gamma}} \hat{\zeta}' \wedge (\hat{v} \sqsubseteq \hat{v}') \wedge (\hat{\sigma}_{k_{\hat{\zeta}}} \sqsubseteq \hat{\sigma}_{k_{\hat{\zeta}'}}) \wedge (\forall \hat{a} \in dom(\hat{\delta}) : \hat{\delta}(\hat{a}) \sqsubseteq \hat{\sigma}_{\hat{\zeta}'}(\hat{a}))$
 $\wedge (\forall \hat{a} \in \hat{\mathcal{R}}_{\hat{v}} \setminus dom(\hat{\delta}) : \hat{\sigma}_{gc}(\hat{a}) \sqsubseteq \hat{\sigma}_{\hat{\zeta}'}(\hat{a})) \wedge (\hat{\mathcal{R}}_{\hat{v}} \subseteq \hat{\mathcal{R}}_{(\hat{\sim}_{\hat{\sigma}_{\hat{\zeta}'}})}(\hat{\mathcal{T}}_{Val}(\hat{v}')))$

The generalisation relaxes the precondition of the implication. That is, the big-step evaluation may use a store other than the small-step one, as long as it is equal to or subsumed by the latter. This is necessary to ensure that the induction hypothesis can be applied in the E-LET case, since the second evaluation step may use a more precise store after continuing with a more precise result from the first evaluation step. Likewise, the generalisation strengthens the postconditions, which now for instance also ensure that the bindings in the resulting delta store are more precise than the corresponding bindings in the resulting store of the small-step evaluation. This is again necessary for the E-LET case, because the second recursive evaluation continues using the delta store produced by the first.

Theorem 11 can now be seen as a corollary of Theorem 12. We elide the proof of Theorem 12 here for brevity; instead, we have verified this property using a mechanised proof⁵, implemented using the Rocq theorem prover.

Discussion. The restriction to λ_{ANF}^- is not arbitrary, as Theorem 11 does not hold in general for λ_{ANF} due to a corner case in the E-LET rule that can arise with strong updates. Consider the expression *let* $x = e_1$ *in* e_2 . After the evaluation of e_1 , the E-LET rule assumes that the bindings in the original store $\hat{\sigma}$ are still necessary to continue with the evaluation of e_2 , and therefore replays the delta store $\hat{\delta}_1$ returned from the evaluation of e_1 on $\hat{\sigma}$. However, if $\hat{\delta}_1$ contains a strong update at an address $\hat{a}$ (i.e., $\hat{a} \in dom(\hat{\delta}_1) \setminus \hat{A}_1$), the old value $\hat{\sigma}(\hat{a})$ is overwritten, so that addresses previously referenced by that value may have become unreachable. If $\hat{\delta}_1$ also contains a new allocation at such an address $\hat{a}'$ (i.e., $\hat{a}' \in dom(\hat{\delta}_1) \cap \hat{A}_1$),

⁵The proof is available online (<https://zenodo.org/records/16731494>) as part of a replication package.

$\widehat{\text{replay}}$ still joins $\hat{\delta}_1(\hat{a}')$ with this old (garbage) value $\hat{\sigma}(\hat{a}')$. In contrast, the small-step interpreter using $(\rightarrow_{\hat{\Gamma}})$ reclaims $\hat{a}'$ immediately after the strong update, so that the new allocation at $\hat{a}'$ is not joined with the old value. Note that this only happens when the evaluation of e_1 strongly updates $\hat{a}$ and subsequently allocates a new value at $\hat{a}'$. We leave a refinement of the $\widehat{\text{E-LET}}$ rule that avoids this issue (e.g., by first applying GC to $\hat{\sigma}$ before applying $\widehat{\text{replay}}$ whenever $\hat{\delta}_1$ contains strong updates), as well as an adaptation of the proof for Theorem 11, as future work.

4.5 Integrating Flow-Sensitive Store Widening

We now formulate a new form of store widening for the big-step semantics using delta stores. Without any form of store widening, a fixpoint computation over the evaluation relation of Section 4.4 might be expensive, or even in some cases not computable. The store widening we propose is similar to the traditional notion of flow sensitivity: it solves the scalability issues, while still allowing strong updates and a limited form of abstract GC. Moreover, it can be efficiently formulated for DSS, as we have previously shown that delta stores are also joinable (which is efficient because of their minimal representation).

Section 4.5.1 explains why store widening is necessary to ensure that program semantics remain computable. Next, Section 4.5.2 shows how the DSS evaluation relation can be modified to integrate such (flow-sensitive) store widening.

4.5.1 The Need for Store Widening

The program semantics for λ_{ANF} expressed in terms of the evaluation relation $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ (Section 4.4) are always computable when $\widehat{\text{Val}}$ is chosen to be finite. The reason is that one can easily show that this assumption renders the relation $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ finite, since both $\widehat{\text{Config}}$ and $\widehat{\text{Result}}$ are also finite. However, two limitations remain:

- The relation can still grow very large, as the size of $\widehat{\text{Store}}$ alone is exponential in the size of $\widehat{\text{Addr}}$ (i.e., in the order of $\mathcal{O}(|\widehat{\text{Val}}|^{|\widehat{\text{Addr}}|})$), potentially leading to many different input configurations that need to be evaluated. In practice, this usually leads to poor scalability towards larger programs, as the time it takes for the abstract interpreter to terminate becomes infeasibly long.
- When choosing an abstract domain $\widehat{\text{Val}}$ that is not finite (but adheres to the ascending chain condition, such as $\mathbf{CP}_{\text{Num}}$), the relation $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ would also no longer be finite, and therefore potentially not computable. In practice, this may lead to the abstract interpreter failing to terminate, preventing decidability. It is reasonable to expect an analysis to be decidable when the abstract domain $\widehat{\text{Val}}$ adheres to the ACC, even if that domain is infinite.

Problem: Abstract GC Inhibits Convergence. Without abstract GC (and in the absence of strong updates), $\widehat{\text{eval}}(e)$ would always be computable for any program e when $\widehat{\text{Val}}$ adheres to the ACC (even when $\widehat{\text{Val}}$ is otherwise infinite). The reason is that the store can only grow over a single execution trace (we say the sequence of stores is *monotonically increasing*), therefore eventually converging (as all values in the store converge, as guaranteed by the ACC). With abstract GC (or strong updates), this no longer holds. Bindings in the store can be removed (or strongly updated to a value that does not subsume the original one), and as such there is no guarantee of convergence in every trace. This requires us to make sure $\widehat{\text{Val}}$ is finite in order to render $\widehat{\text{eval}}(e)$ computable for any e .

We can easily illustrate this using the following Scheme program:

```
(letrec ((f (lambda (n) (f (+ n 1)))))
  (f 0))
```

We assume an abstract interpreter using $\mathbf{CP}_{\text{Num}}$ for the numerical domain and a context-insensitive allocation policy (again writing @n for the address of variable n). The first call to f binds @n to 0. Without abstract GC, the next recursive call to f would extend that binding by joining the existing value with 1, resulting in a configuration where @n is bound to $\top$. Subsequent recursive steps would end up in the same configuration, therefore allowing the fixpoint algorithm to terminate. With abstract GC, the old binding of @n could be garbage collected before the store is extended for the next recursive call. As such, the second call would end up in a configuration where @n is bound to 1, the third one with @n bound to 2 and so on, resulting in an infinite number of configurations to evaluate.

Solution: Store Widening Recovers Convergence. The classical solution to ensure convergence in this setting is to introduce store widening [168]. Traditionally, *global* store widening – as described in Section 2.3.4 – has been applied to both small-step (AAM-based) [168] and big-step (definitional) [40] abstract interpreters. Under global store widening, the stores of all configurations are joined in a single store. This results in a flow-insensitive analysis, as the abstract interpreter does not discern different stores at different program points. Unfortunately, global store widening is incompatible with both abstract GC and abstract counting.

We can, however, choose a middle ground between purely local stores and global store widening. The store widening we propose for DSS in this section joins stores of configurations at the same program point. That is, the analysis keeps track of a single store per program point (being the join of all stores encountered at that program point). This corresponds to a flow-sensitive analysis. While less precise than local stores, it is known that flow sensitivity still allows for strong updates, and as we discuss in Section 4.5.2, a limited form of abstract GC.

4.5.2 Delta Store Semantics with Flow-Sensitive Store Widening

We now show how this form of flow-sensitive store widening can be integrated into the delta store semantics with abstract GC of Section 4.4. Doing so ensures that the evaluation relation $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ becomes finite (and therefore computable) whenever $\widehat{\text{Val}}$ adheres to the ACC. This requires the following two modifications:

- We no longer keep the store directly as part of each configuration. That is, instead of having multiple configurations $\langle e, \hat{\rho}, \hat{\sigma}_i, \hat{t} \rangle$ (i.e., with the same expression e , environment $\hat{\rho}$ and timestamp $\hat{t}$, but different stores $\hat{\sigma}_i$), we now only have a single configuration $\langle e, \hat{\rho}, \hat{t} \rangle$, which gets associated with a single store $\bigsqcup_i \hat{\sigma}_i$. As such, the definition of $\widehat{\text{Config}}$ is updated as follows:

$$\widehat{\text{conf}} \in \widehat{\text{Config}} = \widehat{\text{Exp}} \times \widehat{\text{Env}} \times \widehat{\text{Time}}$$

For a context-insensitive analysis, such a configuration effectively corresponds to a single program point. To associate a single store with each configuration, we make use of a map $\hat{\Xi}_{e_0} : \widehat{\text{Config}} \rightarrow \widehat{\text{Store}}$. The store $\hat{\Xi}_{e_0}(\widehat{\text{conf}})$ over-approximates all stores $\hat{\sigma}_i$ that can occur at the configuration $\widehat{\text{conf}}$ (i.e., that previously were part of that configuration), where e_0 is the initial expression of the program.⁶ We first define a relation $(\rightsquigarrow)_{\widehat{\text{conf}}} \subseteq \widehat{\text{Store}} \times \widehat{\text{Config}}$, where intuitively $\hat{\sigma} \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}$ implies that $\hat{\sigma}$ potentially “flows to” $\widehat{\text{conf}}$ during the evaluation of the configuration $\widehat{\text{conf}}$. We can then define $\hat{\Xi}(\widehat{\text{conf}})$ as the join of all garbage-collected⁷ stores that flow to $\widehat{\text{conf}}$:

$$\hat{\Xi}_{e_0}(\langle e, \hat{\rho}, \hat{t} \rangle) = \bigsqcup \{ \hat{\sigma} \mid \widehat{\mathcal{R}}_{(\rightsquigarrow_{\hat{\sigma}})(\widehat{\mathcal{T}}_{\widehat{\text{Env}}}(\hat{\rho}))} \mid \hat{\sigma} \rightsquigarrow_{\langle e_0, \bigsqcup, \hat{t}_0 \rangle} \langle e, \hat{\rho}, \hat{t} \rangle \}$$

- We render the evaluation relations $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ and $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ deterministic (i.e., they can be seen as functions $\widehat{\text{Config}} \rightarrow \widehat{\text{Result}}$). This is done by joining the results whenever there are multiple non-deterministic program paths.

Together, this suffices to show that $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ becomes a finite relation: it is clear that by factoring out the store, $\widehat{\text{Config}}$ becomes finite, and when $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ is deterministic, only a single result $\widehat{\text{res}} \in \widehat{\text{Result}}$ can be associated with each configuration. And although $(\rightsquigarrow)_{\widehat{\text{conf}}}$ is technically not finite (since an infinite number of stores can flow to a configuration $\widehat{\text{conf}}$ when $\widehat{\text{Val}}$ is infinite), $\hat{\Xi}(\widehat{\text{conf}})$ is computed as the join of all these stores, which is guaranteed to converge due to the ACC.

⁶For brevity, from this point on we often drop the subscript and instead write $\hat{\Xi}$ for $\hat{\Xi}_{e_0}$.

⁷Note that we apply garbage collection before joining the stores, which results in a more precise store compared to doing it the other way around.

We now present the updated evaluation rules for $(\Downarrow^\Delta)$ and $(\Downarrow_{\hat{\Gamma}}^\Delta)$, together with the derivation rules for the relation $(\rightsquigarrow)_{\text{conf}}$ (i.e., included in the same rules).

$$\begin{array}{c}
 \frac{\hat{\sigma} = \hat{\Xi}(\widehat{\text{conf}}) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\underbrace{\langle ae, \hat{\rho}, \hat{t} \rangle}_{\text{conf}} \Downarrow^\Delta \langle \hat{v}, [], \emptyset, \emptyset \rangle} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{\sigma} = \hat{\Xi}(\widehat{\text{conf}}) \quad \hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\delta} = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\underbrace{\langle \text{set } x := ae, \hat{\rho}, \hat{t} \rangle}_{\text{conf}} \Downarrow^\Delta \langle \hat{v}, \hat{\delta}, \emptyset, \{\hat{a}\} \rangle} \quad [\text{E-SET}] \\
 \\
 \frac{\begin{array}{c} \hat{\sigma} = \hat{\Xi}(\widehat{\text{conf}}) \quad (\hat{f}, _) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \\ (\lambda x_i. e_i, \hat{\rho}'_i) \in \hat{f} \quad \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_i = \widehat{\text{alloc}}(x_i, \hat{t}') \\ \hat{\delta}_i = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_i, \hat{v}_x) \quad \hat{\sigma}_i = \hat{\delta}_i \triangleright \hat{\sigma} \end{array}}{\underbrace{\langle e_i, \hat{\rho}'_i[x_i \mapsto \hat{a}_i], \hat{t}' \rangle}_{\text{conf}_i} \Downarrow_{\hat{\Gamma}}^\Delta \langle \hat{v}_i, \hat{\delta}'_i, \hat{A}_i, \hat{U}_i \rangle \quad \hat{\sigma}_{r_i} \rightsquigarrow_{\widehat{\text{conf}}_i} \widehat{\text{conf}}_{r_i}} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{c} \hat{v} = \bigsqcup_i \hat{v}_i \quad \hat{\delta} = \bigsqcup_{\hat{\sigma}} \hat{\delta}'_i \triangleright \hat{\delta}_i \\ \hat{A} = \bigcup_i \{\hat{a}_i\} \cup \hat{A}_i \quad \hat{U} = \bigcup_i \hat{U}_i \end{array}}{\underbrace{\langle f(ae), \hat{\rho}, \hat{t} \rangle}_{\text{conf}} \Downarrow^\Delta \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{c} \hat{\sigma}_{r_i} \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}_{r_i} \quad \hat{\sigma}_i \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}_i \end{array}}{\hat{\sigma} = \hat{\Xi}(\widehat{\text{conf}}) \quad (_, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (_, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \quad \hat{v} = (\emptyset, \hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2))}{\underbrace{\langle op(ae_1, ae_2), \hat{\rho}, \hat{t} \rangle}_{\text{conf}} \Downarrow^\Delta \langle \hat{v}, [], \emptyset, \emptyset \rangle} \quad [\text{E-BINOP}]
 \end{array}$$

$$\begin{array}{c}
 \begin{array}{c}
 \hat{\sigma} = \hat{\Xi}(\widehat{\text{conf}}) \quad \widehat{\text{conf}}_1 \quad \langle e_1, \hat{\rho}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}_x, \hat{\delta}_1, \hat{A}_1, \hat{U}_1 \rangle \quad \hat{\sigma}_{r_1} \rightsquigarrow_{\widehat{\text{conf}}_1} \widehat{\text{conf}}_{r_1} \\
 \hat{\delta}'_1 = \widehat{\text{replay}}(\hat{\delta}_1, \hat{\sigma}, \hat{A}_1) \quad \hat{\sigma}' = \hat{\delta}'_1 \triangleright \hat{\sigma} \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \\
 \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x) \quad \hat{\sigma}'' = \hat{\delta}_x \triangleright \hat{\sigma}' \\
 \widehat{\text{conf}}_2 \quad \langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_2, \hat{A}_2, \hat{U}_2 \rangle \quad \hat{\sigma}_{r_2} \rightsquigarrow_{\widehat{\text{conf}}_2} \widehat{\text{conf}}_{r_2}
 \end{array} \\
 \hline
 \underbrace{\langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{t} \rangle}_{\widehat{\text{conf}}} \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_2 \triangleright \hat{\delta}_x \triangleright \hat{\delta}'_1, \hat{A}_1 \cup \{\hat{a}_x\} \cup \hat{A}_2, \hat{U}_1 \cup \hat{U}_2 \rangle \\
 \begin{array}{c}
 \hat{\sigma} \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}_1 \quad \hat{\sigma}'' \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}_2 \\
 \hat{\sigma}_{r_1} \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}_{r_1} \quad \hat{\sigma}_{r_2} \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}_{r_2}
 \end{array} \\
 \begin{array}{c}
 \hat{\sigma}_{\text{gc}} = \hat{\Xi}(\widehat{\text{conf}}) \quad \langle e, \hat{\rho}, \hat{t} \rangle \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle \\
 \hat{\delta}_{\text{gc}} = \hat{\delta} \upharpoonright_{\widehat{\mathcal{R}}_{(\rightsquigarrow_{\hat{\delta} \triangleright \hat{\sigma}_{\text{gc}}})}(\widehat{\mathcal{T}}_{\text{Val}}(v) \cup \hat{U}_{\text{gc}})} \\
 \hat{A}_{\text{gc}} = \hat{A} \cap \text{dom}(\hat{\delta}_{\text{gc}}) \quad \hat{U}_{\text{gc}} = \{\hat{a} \in \hat{U} \mid \hat{\sigma}_{\text{gc}}(\hat{a}) \neq \perp\} \\
 \underbrace{\langle e, \hat{\rho}, \hat{t} \rangle}_{\widehat{\text{conf}}} \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_{\text{gc}}, \hat{A}_{\text{gc}}, \hat{U}_{\text{gc}} \rangle
 \end{array}
 \end{array} \quad \begin{array}{l} \text{[E-LET]} \\ \\ \text{[E-GC]} \end{array}$$

The first modification is that since the store is no longer part of a configuration $\widehat{\text{conf}}$, we instead now retrieve it as $\hat{\Xi}(\widehat{\text{conf}})$. Conversely, for every configuration $\widehat{\text{conf}}$ (reachable during the evaluation of $\widehat{\text{conf}}$) that normally would have a store component $\hat{\sigma}$, we now instead derive that $\hat{\sigma} \rightsquigarrow_{\widehat{\text{conf}}} \widehat{\text{conf}}$. These derivations come both directly from the evaluation of the current configuration, as well as from recursive evaluations of other configurations. The second modification is that the E-APP rule (the only evaluation rule for λ_{ANF} that previously introduced non-determinism) now joins the results for each possible function that may be called. Doing so makes use of the join operator ($\sqcup_{\hat{\sigma}}$) for delta stores to efficiently merge changes computed w.r.t. $\hat{\sigma}$ over multiple non-deterministic paths.

Note that computing $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ requires a fixpoint computation over both the relations $(\Downarrow_{\hat{\Gamma}}^{\Delta})$ and $(\rightsquigarrow)_{\widehat{\text{conf}}}$. We again refer to the cache-based fixpoint algorithm presented by Darais et al. [40], which can be employed *mutatis mutandis* as one possible implementation for such a fixpoint computation.

4.5.3 Impact on Abstract GC

It is important to reconsider the impact of abstract GC on the interpreter’s precision and performance in the presence of the store widening presented in Section 4.5.2. We recall that abstract GC offers two key advantages (Section 4.1.1):

Avoiding Garbage-Induced Imprecision. Abstract GC avoids precision losses due to (1) unnecessary joins with garbage values, and due to (2) unnecessary increases of their corresponding garbage abstract counts.

1. When using flow-sensitive (“per configuration”) store widening, the precision improvements related to (1) are largely negated. The reason is that addresses are usually identified by the program point they are allocated at, along with the current context (i.e., timestamp). At the same time, for the flow-sensitive store widening presented in Section 4.5.2, all stores sharing the same program point and current context are also joined together. As a result, all values allocated at a certain program point under a certain context are immediately joined together in the same shared store for that program point and context.⁸ For a given address, this directly negates the impact of abstract GC before a fresh allocation at that address.

However, the value bound to an address is not just determined when it is allocated (i.e., when using a combination of alloc and extend), but also when it is updated (i.e., using update, as in the E-SET rule). In these cases, the previous reasoning does not hold, since updates (unlike allocations) for a single address may occur at different program points and contexts, and therefore end up in different stores. Therefore, some precision can still be gained from collecting garbage when such updates occur.

2. The main precision benefit of abstract GC with flow-sensitive store widening is related to (2). With abstract GC, the abstract count of a collected address is reset to **0**, so that it is increased to **1** when the address is allocated again. Store widening will only join (i.e., not *increase*) the abstract counts for all allocations of an address at the same program point and context. Reclaiming garbage bindings before such allocations can therefore keep the count at **1** (since the join of abstract counts **1** and **1** remains **1**) instead of ∞ , increasing the potential for strong updates (and therefore precision).

⁸Note that this assumes that allocations are differentiated using the same context sensitivity as configurations. For less precise allocators (e.g., a monovariant allocator in a context-sensitive analysis) – or when using trace partitioning [99] (which weakens the sharing of stores) – the allocation of an address does not always end up widened in the same store. Therefore, it may be beneficial to reclaim previous occurrences of that address first; such analysis designs can gain more precision when combining flow sensitivity with abstract GC.

Garbage Irrelevance. Abstract GC speeds up the fixpoint computation thanks to garbage irrelevance, and continues to do so in the presence of flow-sensitive store widening. Specifically, garbage irrelevance makes the computation of $\hat{\Xi}$ more efficient, since the fixpoint iteration joining stores at the same program point together reaches a stable value more quickly when garbage bindings are removed. That is, the iteration does not have to continue until these unreachable bindings converge to a stable value as well. In turn, this improves the convergence rate (and memory consumption thanks to smaller stores in $\hat{\Xi}$) for the fixpoint computation.

4.6 Evaluation

We have already formally shown that, for programs in λ_{ANF}^- , DSS with abstract GC always matches or improves upon the precision of abstract GC for a small-step abstract interpreter, using a mechanised proof implemented in Rocq (Section 4.4.3). Likewise, we have discussed the impact of flow sensitivity for DSS, as well as its interaction with abstract GC (Section 4.5.2). We now present an accompanying empirical evaluation, using our implementation in the MAF framework [165].⁹

Specifically, we aim to answer the following research questions:

- RQ1** How much precision is gained using a stackless form of abstract GC (as integrated for DSS in Section 4.4) compared to traditional abstract GC?
- RQ2** What is the impact of flow-sensitive store widening (as discussed in Section 4.5) on the precision and scalability of DSS?
- RQ3** What is the impact of abstract GC for flow-sensitive DSS?

4.6.1 Experimental Setup

We use a similar setup throughout our evaluation.

Environment. We ran all experiments using Scala version 3.1.0 on an Apple MacBook Pro with a 2.3GHz 8-Core Intel Core i9 processor and 16GB of RAM.

Implementation. Our implementation in MAF extends the formalisations for λ_{ANF} to support a large subset of R5RS Scheme. Specifically, we extended MAF with an implementation of DSS (Section 4.3), DSS with abstract GC (Section 4.4) and DSS with abstract GC and flow-sensitive store widening (Section 4.5.2). In addition, we also added implementations of ADI with abstract GC

⁹Although we have not presented a full formal proof for the soundness of DSS, we have validated this empirically using automated soundness testing (as proposed in [7]) in MAF.

(Section 4.2) and a modified AAM implementation with perfect stack precision (using an AAC allocator, see later). Each of these implementations supports various configurations for context sensitivity and the used abstract domains.

Benchmarks. We run our experiments using a benchmark suite of 16 Scheme programs: 10 from the Gabriel benchmarking suite [49] (the same as those used in Chapter 3)¹⁰ and 6 from the built-in benchmarking suite of MAF. Table 4.1 lists each Scheme program along with its size in LOC. We run our experiments using

Benchmark	LOC	Benchmark	LOC	Benchmark	LOC	Benchmark	LOC
boyer	568	deriv	39	takl	18	mceval	282
browse	78	destruc	37	puzzle	144	regex	80
cpstak	19	diviter	8	grid	35	rsa	41
dderiv	83	divrec	11	matrix	648	tak	11

Table 4.1: Overview of the Scheme benchmarks, along with lines of code (LOC) for each benchmark. The Gabriel benchmarks are in **bold**.

m -CFA [109] context sensitivity (which uses the top m stack frames as context¹¹) for varying values of m (where higher values of m may increase precision). For the abstract domain, we employ a constant propagation lattice (Section 2.3.1) as an abstraction for primitive domains in all experiments. As this lattice is not finite, but adheres to the ACC, only DSS-FS is guaranteed to terminate; non-termination for other interpreters results in timeouts. For each of the benchmarks, we use a timeout of 10 minutes (note that benchmarks that timed out are still useful to provide an under-approximation for our precision metric).

Measurements. We compare different abstract interpreters in terms of precision and performance. For precision, we use the built-in precision measurement utilities of the MAF framework. These compare precision by first running the program using a concrete interpreter (multiple times to cover multiple program paths in non-deterministic programs), and subsequently measuring how many values in each abstract interpreter strictly over-approximate (i.e., are less precise than) the direct abstraction of the corresponding results of the concrete interpreter. Specifically, we instrument the concrete interpreter to build up a map $\Psi : Exp \rightarrow$

¹⁰We left out the **ctak** and **triangl** benchmarks of the Gabriel benchmarking suite. The **ctak** benchmark was removed due to its use of **call/cc**, which is not supported by MAF, nor by DSS (as discussed in Section 4.1.1). The **triangl** benchmark was removed because it timed out for the concrete interpreter (and hence could not be used for precision measurements).

¹¹In contrast to k -CFA, which uses the k last call sites. The difference is that the latter also includes call sites of calls that the interpreter already returned from.

$\mathcal{P}(\text{Val})$ that keeps track of all values allocated or returned at every program point. The abstract interpreter similarly builds a map $\hat{\Psi} : \text{Exp} \rightarrow \mathcal{P}(\widehat{\text{Val}})$. We then compute the number of strict over-approximations as $|\{e \in \text{Exp} \mid \alpha(\Psi(e)) \sqsubset \bigsqcup_{\hat{v} \in \hat{\Psi}(e)} \hat{v}\}|$ where $(\mathcal{P}(\text{Val}), \sqsubseteq) \xrightarrow[\alpha]{\gamma} (\widehat{\text{Val}}, \sqsubseteq)$.¹² A more precise analysis should therefore result in fewer over-approximations compared to a less precise one.

4.6.2 RQ1: The Precision Benefit of Stackless Abstract GC

Theorem 11 guarantees that, for programs in λ_{ANF}^- , the precision of DSS with abstract GC (i.e., using the evaluation relation $(\Downarrow_{\hat{\Gamma}}^{\Delta})$) is always the same or better than that of an equivalent small-step abstract interpreter with abstract GC (i.e., using the transition relation $(\widehat{\rightarrow}_{\hat{\Gamma}})$). However, it does not guarantee that the precision is *strictly* better, nor does it tell us *how much* precision improvement can be expected. Moreover, some of the benchmarks in our suite do make use of assignments (through `set!`, `set-car!`, `set-cdr!` and `vector-set!`), and therefore the guarantee of Theorem 11 does not apply to them in general.

In order to evaluate this empirically, we therefore compare precision with an equivalent implementation of AAM with abstract GC and abstract counting (also known as ΓCFA [106]). However, recall that such a small-step abstract interpreter may exhibit decreased precision compared to DSS for two reasons: the lack of stackless abstract GC and the lack of full stack precision (which DSS inherits from its foundation in abstract definitional interpreters). Since we are only interested in measuring the former, we have modified ΓCFA to instead use a fully precise continuation address allocator (specifically, the one from AAC [57]), so that it exhibits the same perfect stack precision as DSS.

Table 4.2 compares the results for ΓCFA and DSS with abstract GC. Note that for benchmarks that timed out, we can use the partial results of the analysis in order to compute a *lower bound* for the number of over-approximations (as the result would only grow more imprecise as the analysis continues).

Comparison of Precision. The results are consistent with Theorem 11 (despite the occasional use of strong updates): in all benchmarks where both ΓCFA and DSS terminate, we have an equal (`cpstak`, `regex`) or lower (`grid`, `rsa`, `tak`) number of over-approximations for DSS. The context-insensitive (i.e., $m = 0$) analyses of the `regex` benchmark also show significant precision improvements for DSS: the analysis with ΓCFA times out with *at least* 34 over-approximations, and requires increased context sensitivity (i.e., $m \geq 1$) in order to achieve the same

¹²The same infrastructure is also used in MAF to perform automated soundness testing. That is, the analysis is considered sound when $\forall e, v \in \Psi(e) : \exists \hat{v} \in \hat{\Psi}(e) : \alpha(\{v\}) \sqsubseteq \hat{v}$.

	$m = 0$				$m = 1$				$m = 2$			
	Γ CFA		DSS		Γ CFA		DSS		Γ CFA		DSS	
	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time
boyer	≥ 0	∞	≥ 0	∞	≥ 0	∞	≥ 0	∞	≥ 0	∞	≥ 0	∞
browse	≥ 5	∞	≥ 7	∞	≥ 0	∞	≥ 4	∞	≥ 7	∞	≥ 4	∞
cpstak	3	50ms	3	26ms	3	11s	3	5s	≥ 3	∞	≥ 0	∞
dderiv	≥ 28	∞	38	20s	≥ 7	∞	7	1s	≥ 11	∞	7	1s
deriv	≥ 3	∞	3	1s	≥ 3	∞	3	1s	≥ 4	∞	3	1s
destruc	≥ 8	∞	≥ 10	∞	≥ 8	∞	≥ 10	∞	≥ 3	∞	≥ 3	∞
diviter	≥ 4	∞	4	3s	≥ 0	∞	4	3s	≥ 0	∞	2	4s
divrec	≥ 4	∞	4	3s	≥ 0	∞	3	3s	≥ 0	∞	3	3s
takl	≥ 6	∞	6	1s	≥ 3	∞	6	9s	≥ 3	∞	6	4m48s
puzzle	≥ 11	∞	≥ 12	∞	11	3m40s	≥ 11	∞	11	4m58s	≥ 5	∞
grid	10	1m34s	7	200ms	≥ 2	∞	7	167ms	≥ 2	∞	7	390ms
matrix	≥ 10	∞	≥ 10	∞	≥ 9	∞	≥ 10	∞	≥ 9	∞	≥ 10	∞
mceval	≥ 3	∞	≥ 3	∞	≥ 3	∞	≥ 3	∞	≥ 12	∞	13	36s
regex	≥ 34	∞	0	92ms	0	293ms	0	95ms	0	275ms	0	100ms
rsa	14	923ms	7	75ms	14	58s	7	92ms	≥ 14	∞	7	78ms
tak	2	20s	0	2s	≥ 0	∞	0	1s	≥ 0	∞	0	1s

Table 4.2: Comparison of the number of strict over-approximations (lower is better) and time taken between small-step Γ CFA and big-step DSS with (stackless) abstract GC. A time of ∞ indicates that the benchmark exceeded the time limit of 10 minutes; in this case, we report a lower bound for the number of over-approximations.

precision as a context-insensitive analysis with DSS. The improvements also hold when context sensitivity is increased for both abstract interpreters: for the **dderiv** benchmark with $m = 2$, Γ CFA has at least 11 over-approximations compared to exactly 7 in DSS. As we have modified Γ CFA with a fully precise continuation allocator, we can attribute these improvements to the stackless nature of abstract GC in DSS, which can reclaim more garbage compared to abstract GC in Γ CFA.

We have also compared precision to the big-step abstract interpreter with abstract GC of Section 4.2, which is more similar to DSS but keeps track of a set of continuation addresses $\hat{\psi}$. Our experiments show identical precision results for this interpreter as for Γ CFA (therefore omitted here for brevity). This confirms that the precision improvements for DSS in Table 4.2 indeed stem from being able to omit $\hat{\psi}$ from the GC root set (i.e., rendering the GC “stackless”).

Comparison of Performance. Both Γ CFA and DSS time out for a significant number of benchmarks, showing poor scalability for both abstract interpreters. Timeouts are more frequent for Γ CFA (the only exception to this being the **puzzle** benchmark for $m = 1$ and $m = 2$). This can partially be explained by its use of the stack-precise AAC continuation allocator [57], which for the context-insensitive case (i.e., $m = 0$) is known to raise analysis complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^8)$.

Further performance improvements can also in part be explained by the increased precision of DSS: as precision increases, the abstract interpreter spends less time having to explore *spurious program paths* (i.e., execution paths that only exist due to excessive over-approximation of the program’s control-flow behaviour). This effect can also be observed for the same abstract interpreter by increasing context sensitivity (e.g., for the `mceval` benchmark with DSS).

4.6.3 RQ2: The Impact of Flow-Sensitive Store Widening

We now evaluate the impact of flow-sensitive store widening on both precision and scalability. For this purpose, we run the same experiments again, this time comparing a version of DSS without store widening (i.e., the $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ evaluation relation of Section 4.4) and a version of DSS with flow-sensitive store widening (DSS-FS) (i.e., using $(\hat{\Downarrow}_{\hat{\Gamma}}^{\Delta})$ of Section 4.5). The results are shown in Table 4.3.

	$m = 0$				$m = 1$				$m = 2$			
	DSS		DSS-FS		DSS		DSS-FS		DSS		DSS-FS	
	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time
boyer	≥ 0	∞	1251	9m46s	≥ 0	∞	≥ 1236	∞	≥ 0	∞	≥ 48	∞
browse	≥ 7	∞	91	8s	≥ 4	∞	88	22s	≥ 4	∞	≥ 80	∞
cpstak	3	26ms	3	9ms	3	5s	3	72ms	≥ 0	∞	3	222ms
dderiv	38	20s	48	905ms	7	1s	46	2s	7	1s	38	5s
deriv	3	1s	29	517ms	3	1s	9	451ms	3	1s	8	787ms
destruc	≥ 10	∞	15	610ms	≥ 10	∞	15	738ms	≥ 3	∞	13	1s
diviter	4	3s	6	1s	4	3s	6	2s	2	4s	4	2s
divrec	4	3s	4	2s	3	3s	4	2s	3	3s	4	2s
takl	6	1s	6	60ms	6	9s	6	143ms	6	4m48s	6	710ms
puzzle	≥ 12	∞	12	429ms	≥ 11	∞	12	885ms	≥ 5	∞	12	2s
grid	7	200ms	12	106ms	7	167ms	12	212ms	7	390ms	10	237ms
matrix	≥ 10	∞	180	8s	≥ 10	∞	129	18s	≥ 10	∞	128	1m27s
mceval	≥ 3	∞	159	19s	≥ 3	∞	157	2m02s	13	36s	≥ 157	∞
regex	0	92ms	48	547ms	0	95ms	48	3s	0	100ms	48	7s
rsa	7	75ms	18	54ms	7	92ms	17	74ms	7	78ms	17	96ms
tak	0	2s	2	5ms	0	1s	2	19ms	0	1s	2	120ms

Table 4.3: Comparison of the number of strict over-approximations (lower is better) and time taken between DSS with and without flow-sensitive store widening (DSS-FS and DSS, resp.). A time of ∞ indicates that the benchmark exceeded the time limit of 10 minutes; in this case, we report a lower bound for the number of over-approximations.

Comparison of Precision. It is clear that store widening, even when applied per configuration, still greatly decreases precision compared to an equivalent abstract interpreter without any store widening. Across all benchmarks, DSS-FS has significantly more over-approximations compared to DSS. A key factor explain-

ing these differences is that abstract GC no longer offers the same precision for DSS-FS as it did for DSS (we explore this further in RQ3). Instead, the analysis is now more reliant on increased context sensitivity in order to improve its precision.

Comparison of Performance. The key benefit of applying store widening is that it greatly improves the performance of the abstract interpreter (and therefore its scalability towards larger programs such as `boyer`, `browse` and `mceval`). Indeed, without widening, many programs fail to terminate for DSS, either because of an exponential explosion in the number of configurations to evaluate or because values in its (infinite) domain can never converge. In contrast, for DSS-FS, we only observe 4 timeouts in total, and in fact none for the context-insensitive variant (i.e., where $m = 0$). In other benchmarks (e.g., `tak1` with $m = 2$), performance is improved by orders of magnitude. While the precision benefits of DSS without store widening are appealing, we therefore argue that *some* store widening (such as flow-sensitive store widening) is a must to analyse larger, real-world programs.

4.6.4 RQ3: The Impact of Abstract GC on Flow-Sensitive DSS

As discussed in Section 4.5.2, store widening limits the precision benefits of abstract GC. We can, however, expect it to have a positive impact on the performance of the analysis by reducing the number of iterations that are required for each flow-sensitive store to converge. We evaluate the impact of abstract GC in the setting of flow-sensitive store widening by comparing two versions of DSS-FS: one with and one without abstract GC. Table 4.4 shows the results.

Comparison of Precision. As expected, abstract GC has very little impact on precision when employing flow-sensitive store widening. The only benchmarks that show some minor precision improvement are `browse` and `matrix`. The reasons for these limited precision improvements are discussed in Section 4.5.2. In summary, abstract GC’s main precision benefit comes from updates (i.e., updates to mutable variables and mutable data structures). However, these may not occur frequently in our benchmarking suite: although Scheme is technically an imperative language, in practice it often encourages a more functional style, avoiding such side-effecting mutations. The benefits on DSS-FS with abstract GC may therefore be more pronounced for languages with programming styles that make frequent use of (field) assignments (such as Python or JavaScript).

Comparison of Performance. Both configurations achieve comparable performance results, with some benchmarks (such as `boyer`) showing a clear edge for

	$m = 0$				$m = 1$				$m = 2$			
	with GC		without GC		with GC		without GC		with GC		without GC	
	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time	imprec	time
boyer	1251	9m46s	1251	3m12s	≥ 1236	∞	1249	6m13s	≥ 48	∞	48	1m36s
browse	91	8s	98	2s	88	22s	88	8s	≥ 80	∞	80	7m33s
cpstak	3	9ms	3	14ms	3	72ms	3	50ms	3	222ms	3	129ms
dderiv	48	905ms	48	379ms	46	2s	46	1s	38	5s	38	4s
deriv	29	517ms	29	223ms	9	451ms	9	288ms	8	787ms	8	453ms
destruc	15	610ms	15	449ms	15	738ms	15	743ms	13	1s	13	1s
diviter	6	1s	6	2s	6	2s	6	1s	4	2s	4	1s
divrec	4	2s	4	2s	4	2s	4	1s	4	2s	4	1s
takl	6	60ms	6	45ms	6	143ms	6	100ms	6	710ms	6	908ms
puzzle	12	429ms	12	173ms	12	885ms	12	901ms	12	2s	12	4s
grid	12	106ms	12	185ms	12	212ms	12	301ms	10	237ms	10	374ms
matrix	180	8s	181	7s	129	18s	129	32s	128	1m27s	128	3m50s
mceval	159	19s	159	7s	157	2m02s	157	1m27s	≥ 157	∞	≥ 151	∞
regex	48	547ms	48	387ms	48	3s	48	3s	48	7s	48	12s
rsa	18	54ms	18	112ms	17	74ms	17	90ms	17	96ms	17	158ms
tak	2	5ms	2	6ms	2	19ms	2	28ms	2	120ms	2	323ms

Table 4.4: Comparison of the number of strict over-approximations (lower is better) and time taken between DSS-FS with and without abstract GC. A time of ∞ indicates that the benchmark exceeded the time limit of 10 minutes; in this case, we report a lower bound for the number of over-approximations.

DSS-FS without GC, and others (such as **matrix**) exhibiting better performance using DSS-FS with GC. The reason is that the integration of abstract GC impacts performance both positively and negatively.

The negative impact comes from the overhead associated with GC. In our implementation specifically, we apply a tracing stop-and-copy GC at every evaluation step (in order to maximise the precision benefits); we have previously shown that such a policy can potentially slow down an abstract interpreter by one or two orders of magnitude (cf. Chapter 3), and is therefore not recommended in practice. Therefore, we argue that future work should explore the integration of abstract reference counting as an alternative for tracing GC in DSS. Doing so could overcome this significant performance overhead and potentially allow for a more consistent net positive performance impact from abstract GC.

Despite this significant overhead, the performance still holds up well compared to DSS-FS without abstract GC. This can be attributed to the positive impact of abstract GC on performance, as the stores at each program point converge faster without garbage. To measure this benefit without the technical overhead of abstract GC, we conducted a separate experiment measuring the number of iteration steps required for the fixpoint computation of the analysis. For instance,

the `matrix` benchmark only requires 2448 iterations with GC compared to 3098 without, explaining its performance improvement. On average, for our experiments abstract GC decreases the number of iterations required by 27.8%.

4.7 Related Work

We refer to Section 3.5 for an overview of related work on abstract GC and abstract counting, and their integration for small-step (AAM-like) abstract interpreters. In this section, we focus on related work on big-step abstract (definitional) interpreters (pushdown control-flow analyses in general), the integration of abstract GC for such interpreters, and their support for flow and control sensitivity.

Pushdown Control-Flow Analyses. Several approaches have been formulated to model abstract interpreters as pushdown systems, which offer an unbounded stack and therefore also *perfect stack precision* (i.e., perfect matching of calls and returns, thereby avoiding imprecision in the interpreters’ control-flow behaviour).

DSS achieves the pushdown control-flow property by following the approach of Darais et al. [40] to construct big-step abstract (definitional) interpreters. Abstract definitional interpreters are based on Reynolds’ (concrete) definitional interpreters [132], which inherit the semantics of the defining (meta)language in order to define the semantics for the defined language. The key point is that abstract definitional interpreters — and therefore also DSS — obtain the pushdown control-flow property “for free” by inheriting the matching of calls and returns from the defining (meta)language. This avoids the need for an explicit abstraction of the continuation and results in perfect stack precision.

The small-step (AAM-style) abstract interpreters of Chapter 3 and the big-step (ADI-style) abstract interpreters of this chapter can systematically be derived from one another. Wei et al. [177] identify a series of well-known transformations that can turn one into another. The AAM-to-ADI transformation involves *refunctionalising* (i.e., replacing data types representing continuations with higher-order functions) explicit continuations. The authors show that ADI is obtained from the refunctionalisation of an AAM machine with an unbounded stack (and hence perfect stack precision), thereby confirming the perfect stack precision of ADI. The ADI-to-AAM transformation follows the inverse for each of the steps (e.g., *defunctionalisation* instead of refunctionalisation) in the opposite order.

Earlier work by Schmidt [141, 142] already showed how abstract interpreters can be derived directly from a language’s big-step (“natural”) semantics. In their approach, an (abstract) interpretation of a program is a derivation tree over the language’s big-step evaluation rules. In retrospect, this approach already exhibits

the exact call-return matching of ADI and DSS (i.e., “perfect stack precision”), as the result of a callee flows only to its caller node in the derivation tree. In contrast to our own work, Schmidt’s abstract interpreter follows a co-inductive interpretation of the big-step evaluation rules to allow for infinite derivation trees that model looping behaviour.¹³ DSS instead uses an inductive interpretation by defining $(\Downarrow^\Delta)$ as the minimal relation closed over the big-step evaluation rules. As such, an infinitely looping program has no derivation under $(\Downarrow^\Delta)$, whereas Schmidt allows an infinite derivation tree for such diverging programs. In Chapter 5, we formulate a fixpoint algorithm that also computes a total version of $(\Downarrow^\Delta)$.

The term *pushdown control-flow analysis* (PDCFA) comes from the work of Earl et al. [44]. They point out that an abstraction of the concrete state space Σ does not necessarily need to be finite (which is the case for AAM and the abstract interpreters of Chapter 3), as this (unnecessarily) causes too much imprecision in the matching of calls and returns. Instead, PDCFA abstracts the state space into a pushdown automaton and formulates the analysis in terms of certain pushdown reachability, which is a decidable property for pushdown automata. Compared to the ADI approach (and DSS), the cost of PDCFA is prohibitive ($\mathcal{O}(n^6)$ instead of $\mathcal{O}(n^3)$ for a monovariant analysis) and requires significant engineering efforts.

CFA2 [171] uses a small-step abstract interpreter with an unbounded continuation (i.e., a stack of unbounded height, without approximations) to obtain perfect stack precision. Due to the unbounded stack, the abstract state space becomes infinite. CFA2 therefore employs a summarisation algorithm [131] to analyse this state space (treating the analysis as a graph reachability problem) while preserving termination. A later version of CFA2, named Big CFA2 [170], instead obtains perfect stack precision by trading the unbounded stack for a big-step abstract interpreter. The author points out that Big CFA2 enjoys improved scalability over (small-step) CFA2 at the cost of not supporting first-class control operations (due to the lack of an explicit continuation), as is also the case for DSS.

Other approaches aim to achieve the pushdown control-flow property directly for AAM-based abstract interpreters. Glaze et al. [57] show that AAM can achieve perfect stack precision by using a continuation allocator (i.e., $\widehat{\text{alloc}}_k$) that is precise enough to avoid any mismatching of calls and returns. Unfortunately, their resulting analysis, dubbed *abstracting abstract control* (AAC), significantly increases the complexity of the analysis ($\mathcal{O}(n^8)$ instead of $\mathcal{O}(n^3)$ for a monovariant analysis). Gilray et al. [56] later point out that the same perfect stack precision can be obtained through a simpler continuation allocator, which constructs continuation addresses consisting solely of the target expression and environment. They show

¹³Note that under abstract interpretation, looping behaviour is more common — e.g., a finite sequence of iterations in a concrete execution is typically folded into a loop in the abstract.

that this allocator does not increase the analysis complexity (i.e., a monovariant analysis remains $\mathcal{O}(n^3)$), yet achieves perfect stack precision (they refer to the resulting analysis as P4F – *pushdown CFA for free*). However, an important limitation of P4F is that it only achieves perfect stack precision for free when using global store widening. Germane et al. [51] also point out that an m -CFA allocation policy (as we have used throughout this dissertation) achieves perfect stack precision for free in the same sense as P4F (with the same limitation that it requires global store widening). An m -CFA timestamp summarises the top m stack frames (instead of k -CFA’s last k call sites) and rebinds addresses in the environment to achieve polynomial instead of exponential time complexity [109]. The former ensures that both the timestamp and environment remain constant throughout an abstract invocation; the latter ensures that the environment becomes isomorphic to the timestamp, so that m -CFA’s continuation addresses uniquely identify the invocation entry configuration (the P4F invariant that yields full stack precision).

Pushdown Control-Flow Analyses with Abstract GC. Existing work has made several attempts to integrate abstract GC into abstract interpreters that enjoy the benefits of pushdown control flow, such as the big-step abstract definitional interpreters of Darais et al. [40], aiming to get the benefits of both techniques. We discussed the original integration of abstract GC into abstract definitional interpreters, and the shortcomings of this integration, in Section 4.2.

Earl et al. [45] show how abstract GC can be integrated into their pushdown control-flow analysis. They formulate their approach as *stack introspection*: for every control point they approximate the set of continuation frames that could be on the stack, using their references as part of the root set for abstract GC. Similar to the suboptimal integration of abstract GC into abstract definitional interpreters, the resulting analysis no longer enjoys the context irrelevance of the original pushdown analysis, and does not achieve the same potential in precision improvements of abstract GC as the stackless form of abstract GC for DSS.

CFA2 [171] combines full stack precision with some form of abstract GC. CFA2 separates bindings on the stack from those on the heap: a reference is a stack reference if it appears at the same nesting level as its binder; otherwise, the reference is allocated on the heap. Stack bindings are automatically “garbage collected” as stack frames are popped; this is similar to how DSS can garbage collect (unreferenced) local variables (regardless of whether they are already allocated for the continuation) in a delta store upon returning. Stack bindings also support strong updates and *stack filtering*; the latter improves precision by narrowing a stack binding to only a single closure, if that closure was previously selected as the operator for a function call. DSS instead supports strong updates

more generally for all bindings through abstract counting, and could similarly support the stack filtering optimisation for heap bindings for which the abstract count is **1**. For bindings that escape to the heap, CFA2 does not provide any GC or strong updates. The authors note that techniques such as abstract GC and abstract counting could be added on top of CFA2 for heap-allocated bindings.

Monat et al. [112] integrate abstract GC into a flow-sensitive analysis for Python (in the MOPSA framework). In contrast to our own work, their abstract garbage collector still keeps continuation references alive, and therefore does not benefit from the precision improvements of “stackless” abstract GC. Similar to our own findings on RQ3 (cf. Section 4.6.4), they also report limited precision benefits when flow-sensitive widening is used, and argue that abstract GC is mainly useful to improve analysis performance and memory consumption. They also point out that abstract GC *may* improve precision due to its interaction with recency abstraction [13]. In Section 4.5, we also argued that abstract GC may still improve precision due to its interaction with abstract counting and (strong) updates.

The closest to our own work is the *heap fragment* semantics (HFAC) of Germane et al. [53, 54]. Similar to how delta stores in DSS represent changes w.r.t. an original store, a heap fragment in HFAC only captures bindings relevant for the current evaluation. To our knowledge, HFAC is the only other existing formalism exhibiting all advantages of pushdown control flow and abstract GC, as listed in Section 4.1.1. We improve upon the work of [53] by adding support for abstract counting and collecting garbage in delta stores (whereas the technique presented in [53] only collected garbage for the input heap fragment). The authors of [54] point out the “formal weight” of HFAC, which significantly complicates the language semantics. Compared to their work, we present a novel and simpler formalism, capturing the essence of the mechanism that allows the integration of abstract GC. We also show how delta stores can be joined, and extend the formalism to incorporate a novel form of store widening that preserves flow sensitivity.

Control and Flow Sensitivity. Darais et al. [39] introduce *Galois transformers* as composable analysis building blocks that each control analysis parameters (such as context sensitivity). By varying the Galois transformer that controls the analysis’ handling of nondeterminism (and reordering the Galois transformers of a monadic interpreter), they identify three different variants of path and flow sensitivity, so that an analysis is either *path-sensitive*, *flow-sensitive* or *flow-insensitive*. In more recent work, Germane et al. [52] make the same categorisation for different kinds of “control-flow sensitivity”. A path-sensitive analysis keeps abstract states with different analysis results (including their store) separate (e.g., AAM with local stores), resulting in maximal precision but poor scalability (due to a

potential explosion in the number of abstract states). DSS without store widening (as presented in Section 4.3) is an example of such a path-sensitive analysis. A flow-sensitive analysis merges the analysis results per program point, following our own interpretation of flow sensitivity as formulated for DSS with store widening in Section 4.5. Flow sensitivity has so far been more common in traditional data flow analyses [62, 115, 66]. A flow-insensitive analysis merges (for each variable) all analysis results for the entire program, and corresponds to the global store widening presented in Section 2.3.4. Interestingly, Darais et al. also provide a single, generic implementation of abstract GC (i.e., implemented against a monadic interface) that can be used for interpreters using varying path and flow sensitivities. However, unlike DSS, they integrate abstract GC for a small-step abstract interpreter, and always include the continuation addresses as part of the GC root set (therefore missing out on the advantages of stackless GC).

The big-step abstract interpreters of Schmidt [141, 142] can be considered flow sensitive as well. Schmidt’s interpreters guarantee termination through *trace memoisation*: whenever the same program point recurs during a(n abstract) derivation, it reuses the existing derivation for that program point (i.e., each program point appears at most once in a derivation tree). In doing so, the new inputs (i.e., environment/heap bindings) are joined with the existing inputs in the tree, similar to how DSS with flow-sensitive store widening joins the input stores per program point. Trace memoisation shares the same requirement as flow-sensitive store widening on the abstract domain: the ascending chain condition is required in order to guarantee that all inputs in the derivation tree eventually converge.

Hardekopf et al. [63] propose parameterising an analysis with a widening operator on abstract states. The choice of this widening operator determines how abstract states are partitioned — i.e., how different abstract states are joined together to guarantee termination and improve analysis scalability. They show that both context sensitivity and flow sensitivity can be controlled by varying choices of this operator. However, this operator is not immediately applicable to big-step abstract interpreters (such as those underlying DSS), as it requires access to some form of an explicit continuation in order to control control-flow behaviour. In contrast, it could directly be applied to the small-step abstract interpreters of Chapters 2 and 3 to parameterise both context sensitivity and flow sensitivity.

In their work on combinator-based fixpoint algorithms for big-step abstract interpreters (such as DSS), Keidel et al. [84] propose *stack widening* to avoid exploring an infinite chain of input configurations. Stack widening uses a widening

operator¹⁴ on the input environment¹⁵ whenever the same program point recurs during its own evaluation (i.e., when the encountered program point is already “on the stack”). As such, stack widening can be seen as a less aggressive form of flow-sensitive store widening: the former only joins the input stores for those configurations at the same program point that occur in the same loop, whereas the latter joins the input stores for *all* configurations at the same program point. Stack widening therefore retains more precision than flow-sensitive store widening, but may also be less scalable (as it keeps more configurations distinct). For abstract GC, stack widening presents an interesting alternative to flow-sensitive store widening: an address allocated at the same program point is no longer guaranteed to be joined with all other allocations at that program point (only with those that occur in the same loop). Therefore, reclaiming old addresses using abstract GC may improve precision more significantly, overcoming some of the limitations of flow-sensitive store widening (discussed in Section 4.5.3), while still avoiding the same issues that motivated its use (Section 4.5.1). We leave the exploration of stack widening for DSS with abstract GC as future work.

The flow-sensitive store widening of Section 4.5 joins stores per control location (and context). *Trace partitioning* [99, 113] can generalise this technique by using abstract traces (i.e., information based on the history of the interpreter’s execution) to keep multiple stores at the same control location (but with different execution traces reaching that control location) separate. Keeping more stores separate improves precision, and may also increase the precision benefits of abstract GC (compared to what we observed in RQ3) as different values allocated at the same address may be joined in different stores (therefore making it useful to reclaim previous bindings at that address using abstract GC).

Interestingly, Oh et al. [124] suggest an *adaptive* approach to control-flow sensitivity, where different addresses are treated with different sensitivities (in their work, only handling them either flow-sensitively or flow-insensitively). For our own experiments, it is clear that the precision benefits of path sensitivity in combination with abstract GC are substantial, but impede analysis scalability when applied to all addresses. Applying their learning strategy for choosing between both control-flow sensitivities adaptively for each address could potentially bring together the benefits of both DSS and DSS-FS.

In order to improve the scalability of flow-sensitive analyses, Oh et al. [123] present a general framework for *sparse analysis* techniques for abstract interpreters. In a traditional flow-sensitive analysis (as in DSS-FS), the abstract state (i.e.,

¹⁴In our setting, we can replace this widening operator with the join operator ($\sqcup$), as all domains adhere to the ascending chain condition.

¹⁵In our setting, this corresponds to the store $\hat{\sigma}_\xi$ of an input configuration ξ .

store in DSS-FS) associated with a program point is propagated to the successor program points. A binding is often just “passed through” many program points, and only used occasionally by other program points later on. In a sparse analysis, such bindings can directly be propagated from the definition sites to the program points where they are actually used. That is, in such a sparse analysis bindings get propagated directly following *data dependencies* instead of being passed along through the control-flow paths of the program. These data dependencies can be computed through a (cheap) pre-analysis. In contrast to abstract GC, using such a sparse analysis optimisation does not impact precision at all. Instead, it improves scalability, as updated bindings do not need to be propagated through intermediary states that do not make use of these bindings. As such, it provides a form of garbage irrelevance (based on usage instead of reachability) by avoiding the need to revisit those states upon a change of an unused binding, and reduces memory consumption per program point. Similar to DSS-FS, a sparse analysis can limit the store of a program point to only include relevant bindings. The difference is that for DSS-FS, these relevant bindings also include bindings that only need to be propagated. That is, unlike sparse analyses, DSS-FS propagates the entire (garbage-collected) store to the successor configurations (i.e., following the control-flow of the program). We leave a sparse formulation of DSS-FS to further improve its scalability open as future work. Note, however, that an incorporation of reachability-based abstract GC into such a formulation may harm sparsity, as computing reachability needs to use the references of all reachable addresses.

4.8 Conclusion

In this work, we have presented delta store semantics (DSS), a novel formulation for big-step abstract definitional interpreters where evaluation steps return a delta store to capture all changes made to the input store. Unlike regular stores, delta stores are more minimal (representing only changes), reusable (being able to be applied to or *replayed* for a (larger) store), composable, and efficiently joinable. We have shown how these delta stores enable the integration of both abstract GC and a flow-sensitive variant of store widening into DSS.

When using DSS with abstract GC — without store widening — we not only achieve the full advantages of pushdown control flow and abstract GC, but also unlock an additional synergy between both that further increases the benefits of abstract GC. Specifically, we have shown both formally (using the Rocq theorem prover, for programs without variable assignment) and empirically (using our implementation in MAF, including programs with assignments) that this combination outperforms a small-step interpreter with abstract GC.

When using DSS with flow-sensitive store widening, our experiments confirm that DSS no longer faces the scalability issues that are inherent to the usage of local (path-sensitive) stores. Unlike global (flow-insensitive) store widening, we have shown that the resulting abstract interpreter still supports abstract counting (i.e., strong updates) and remains compatible with abstract GC.

5 A Parallel Worklist Algorithm for DSS

In Chapter 4, we presented a formal description of a big-step (abstract) semantics with abstract GC. This description was mostly declarative: while we defined the program semantics using an evaluation relation with recursive evaluation rules, we left open *how* a fixpoint for this recursive relation should be computed operationally. In this chapter, we propose the use of an efficient, parallel worklist algorithm to compute the program semantics for DSS. As a baseline, we generalise an existing *effect-driven* worklist algorithm (originally designed for a class of modular, flow-insensitive analyses), so that it can also be instantiated for DSS.

Next, we focus on an efficient parallelisation of this worklist algorithm, taking advantage of today’s multi-core CPUs to further speed up the underlying analysis. Existing work on parallel analysis usually targets traditional data flow analyses for static, first-order languages (such as C), where dependencies between analysis results are known a priori (simplifying parallelisation strategies). In contrast, our parallel worklist algorithm can handle more complex analyses for “harder-to-analyse” higher-order, dynamic languages (such as JavaScript), where dependencies between analysis results are only discovered during the analysis itself. The key insight of our parallelisation technique is to exploit the *modular* nature of the effect-driven worklist algorithm, in which different *components* can be analysed in isolation from one another (and therefore also in parallel). The parallel worklist algorithm takes into account dependencies between these components to correctly orchestrate their analyses. At the same time, we can show that it can further take advantage of the monotonicity of the analysis to speed up its convergence.

While we instantiate our parallel worklist algorithm for DSS, it remains applicable and efficient for a large class of analyses where dependencies need to be computed during the analysis itself. We demonstrate this by evaluating our parallel algorithm for another existing analysis, MODF. Using a set of Scheme benchmarks, we observe good scalability of our parallel worklist algorithm for DSS(-FS) and MODF up to 16 workers (with most speedups in the $4\times$ – $16\times$ range).

5.1 Motivation

In Chapter 4, we defined the abstract semantics for λ_{ANF} in terms of the evaluation relations $(\Downarrow^\Delta)$ and $(\Downarrow_{\text{F}}^\Delta)$. We defined $(\Downarrow^\Delta)$ and $(\Downarrow_{\text{F}}^\Delta)$ as the minimal relations closed over a given set of evaluation rules. In theory, one can directly apply a Kleene-Tarski iteration [157, 87] to compute these minimal relations as the least fixpoint. However, such a “bottom-up” computation is not practical, as all possible configurations (for a given program) in $\widehat{\text{Config}}$ would have to be enumerated and evaluated. In the original work on abstract definitional interpreters, Darais et al. [40] describe a “top-down” cache-based fixpoint algorithm to compute the abstract semantics. Such a top-down approach starts from the topmost expression of the program to analyse, and only explores configurations that are reachable from this expression through recursive evaluations. However, this algorithm can still not be considered efficient: every iteration needs to re-evaluate the entire program, based on the results of the previous iteration. The algorithm stops when no new results are discovered with respect to the previous iteration.

In contrast, as the foundation for this chapter, we employ an *effect-driven* worklist algorithm, originally designed by Nicolay et al. [119] in the context of modular program analysis. This algorithm employs a worklist of *components* that need to be analysed (i.e., configurations for which the result needs to be computed). It also keeps track of *dependencies* between the analyses of components (i.e., derivations of the evaluation), so that whenever information is updated (i.e., new derivations are discovered), components that are impacted by this updated information are added back to the worklist for (re-)analysis (we say an *effect* is triggered). The algorithm terminates when the worklist is empty (i.e., no more effects are discovered). This approach is generally more efficient, as it avoids having to re-evaluate the entire program on each iteration.

In this chapter, we aim to further improve the performance of this worklist algorithm through parallelisation. The motivation is twofold. First, we identify opportunities for parallelisation inherent to this algorithm. Therefore, we can speed up the program analysis *for free* (i.e., by exploiting these existing opportunities). Second, a parallelisation of this worklist algorithm enables the parallelisation of a large class of analyses where dependencies between analysis results cannot be known upfront (e.g., for higher-order languages). Existing work has shown that such languages are harder to analyse (see Section 5.1.1). We can automatically render these analyses parallel *for free* (i.e., without any modifications to the underlying analysis itself) by swapping the existing sequential worklist algorithm with our parallel algorithm. The only requirement is that these analyses can reify their computational dependencies.

5.1.1 Problem: Parallel Analysis for Higher-Order Languages

In order to be useful, static analyses require both good precision and performance. High precision can be achieved through various techniques, such as increasing context sensitivity [146, 55] or using a more precise abstract domain [33]. Unfortunately, such precision-increasing techniques often come at the cost of increasing the complexity of the analysis, therefore also impacting its performance. Abstract GC is one of the few techniques that usually improves both precision and performance. Another “free” approach to speed up the analysis is to exploit today’s ubiquitous multi-core CPU architectures and *parallelise* the analysis.

There is ample existing work on developing such parallel analyses. However, most of the existing parallel analysers target rather static languages such as C [114, 181, 100, 86, 4]. An advantage when parallelising the analysis for such languages is that the control-flow dependencies of the analysed program are almost entirely known beforehand (i.e., the *inter-procedural call graph* is available a priori). For instance, the SATURN analyser [181] exploits these call dependencies by parallelising a *bottom-up analysis*, in which a function is only analysed after all its callees have been analysed. Such a *bottom-up analysis* is almost trivially parallelisable: the analysis can start by analysing all functions at the bottom of the call dependency graph (those without any callees) in parallel, then analyse all subsequent functions whose callees have already been analysed in parallel, and so on¹. The authors report speedups up to $29\times$ (for 80 cores) using this approach.

In contrast, less attention has been given so far to the parallel analysis of highly dynamic, higher-order languages such as JavaScript or Scheme. Parallelising an analysis for these languages is more challenging, as the control-flow behaviour of the program and dependencies between analysis results are not known beforehand, and only discovered during the analysis itself. The parallelisation strategies described previously would for instance not be applicable to DSS (Chapter 4), as DSS interleaves the computation of the program’s control and data flow (e.g., the program path chosen by the E-APP rule is determined by the set of closures flowing to the operator value). Other parallelisation strategies that can handle such analyses fall short in their results and applicability. For instance, Dewey et al. [42] parallelise JSAI, an abstract interpreter for JavaScript, by exploring multiple independent program states in the analysis in parallel. Compared to parallel analyses for static languages, the speedups are more modest here: on 12 cores, most benchmarks achieve a $2\text{--}4\times$ speedup. Furthermore, as the exploration of

¹Note that in such a bottom-up analysis, functions that are (mutually) recursive form a strongly connected component (SCC) in the call graph, and need to be analysed together in a single fixpoint computation.

program states is parallelised by context, their approach is only able to parallelise context-sensitive analyses, and its efficiency is very dependent upon the choice and impact of context sensitivity for the analysed program.

5.1.2 Approach: A Parallel, Effect-Driven Worklist Algorithm

In this chapter, we present a novel approach to automatically parallelise a general class of analysers for dynamic, higher-order languages. Specifically, we propose the parallelisation of an effect-driven worklist algorithm for *modular analyses*. In a modular analysis [35], a program is split up into different components (such as the functions in the program), and those components are repeatedly analysed in isolation. For DSS, when expressed as a modular analysis, these components correspond to configurations that are to be evaluated.

We consider a modern formulation of modular analyses for highly dynamic and higher-order languages, using an effect-driven worklist algorithm that can keep track of dependencies discovered during the analysis itself. This worklist algorithm has recently been used for both inter-procedural analysis [119] and for the inter-process analysis of concurrent programs [153]. Although these analyses have been touted for their applicability and scalability, so far no attention has been given to their inherent opportunities for parallelisation. Concretely, we identify the following opportunities for an efficient parallelisation of this worklist algorithm:

- The key insight is that the modularity of the analysis can be exploited for parallelisation: as the analysis of a single component is done in isolation, multiple components can safely be analysed in parallel. In order to ensure that the overhead of the parallel scheduling does not overshadow the potential for speedups, it is important to use sufficiently coarse-grained components (i.e., components that take long enough to analyse individually).
- The worklist algorithm is effect-driven. Therefore, computational dependencies between the analyses of different components are already reified (since these are needed to determine which components are impacted when an effect is triggered). The availability of these reified dependencies enables determining if parallel analyses of multiple components are truly independent of one another. If not, the parallel algorithm can correctly take into account any possible interference on shared analysis state.
- A final insight is that we can exploit the monotonicity of the analysis. The analysis of a component may no longer be up-to-date if a parallel analysis of another component triggered an effect that updated one of its dependencies. In general, a parallelisation technique such as Software Transactional Memory (STM) [145] would therefore invalidate the result computed for

that component. However, in this specific domain, due to the monotonicity of the analysis, we are certain that this updated state for this dependency will only lead to results that subsume the outdated information. As such, the effects computed by an outdated analysis of a component can still be used to update the analysis state to speed up its convergence. This insight is not novel: our observation had already been formally stated in an early theoretical framework for parallel fixpoint computations by Cousot [32].

We first generalise the sequential worklist algorithm of Nicolay et al. (Section 5.2), so that it can also be instantiated for DSS and DSS-FS. Next, we formulate a new parallel variant of this worklist algorithm (Section 5.3) that yields the same results as the sequential version. The resulting algorithm remains completely agnostic of the particular instantiation of the modular analysis. While we focus on DSS, it can be applied to render any existing modular analysis parallel *for free*.

5.2 A Sequential Worklist Algorithm for DSS

We first consider a sequential version of the effect-driven worklist algorithm. Nicolay et al. [119] originally designed this algorithm for a flow-insensitive analysis (i.e., an analysis using a global store) without any form of abstract counting and abstract GC. In this section, we show how this worklist algorithm can be generalised to support more analyses (Section 5.2.1), and then demonstrate this by instantiating it for both DSS (Section 5.2.2) and DSS-FS (Section 5.2.3).

5.2.1 Effect-Driven Worklist Algorithm

The core idea of a modular analysis is that the program under analysis can be divided into multiple *components*, which are analysed separately and for which the results can be combined to synthesise the analysis result for the whole program [35]. The exact definition of a component can be chosen depending on the given program and the goal of the analysis, and typically represents an abstraction of some run-time entities in the program (such as function calls [119] or threads [153]). Ideally, these components can be analysed independently of one another, enabling a “divide-and-conquer” approach to program analysis. However, usually analyses of components can still influence and depend on one another. For instance, when components are chosen to be functions, the analysis of one function may modify a global variable read by another, requiring a re-analysis of the latter. Similarly, when components are chosen to be threads, the analysis of one

thread may spawn a new thread that needs to be analysed. In general, as the analyses of components are interdependent, some form of fixpoint computation is necessary to analyse components multiple times until convergence is reached.

We consider an *effect-driven* worklist algorithm to efficiently compute this fixpoint. The algorithm orchestrates the analysis of components in a modular analysis. The key idea is that a component only needs to be (re-)analysed when it has potentially been impacted by the analysis of another component. Therefore, the algorithm keeps track of the *dependencies* of each component, and schedules a component c for re-analysis as soon as the analysis of another component triggers an *effect* that impacts one of the dependencies of component c . Concretely, each effect modifies some part of the global analysis state, and may trigger re-analyses (using the updated analysis state) for components that had dependencies on that part of the analysis state. The algorithm terminates when no more effects are triggered, indicating that the global analysis state has converged.

The original version of this worklist algorithm [119]² was formulated for a modular analysis where components correspond to function calls and where the analysis state corresponds to a single global store (since the analysis is flow-insensitive). We present a generalised version of this algorithm, leaving open the definition of components, the global analysis state and the analysis of components as configurable parameters. Different choices for these parameters lead to different instantiations of modular analyses (we show one for DSS and DSS-FS in Sections 5.2.2 and 5.2.3, respectively). Concretely, an instantiation of the analysis should provide:

- A choice of $\widehat{\text{Component}}$ to represent components. For example, in a function-modular analysis, $\widehat{\text{Component}}$ is the set of all functions in the program.³ The instantiation should also provide some component $c_0 \in \widehat{\text{Component}}$ that represents the “entry point” of the program. In a function-modular analysis, c_0 would represent the initial call to the `main` function of the program.
- A choice of $\widehat{\text{Dependency}}$ to represent dependencies. Dependencies identify a part of the global analysis state that the analysis of a component may depend on. For example, in a flow-insensitive analysis, the global analysis state contains the global store of the analysis. A dependency then corresponds to an address in the shared global store. Typically, this analysis state also contains a *cache* holding the current analysis results per component. Mutually dependent component analyses can register dependencies on each other’s results, leaving the fixpoint computation to Algorithm 3.

²Technically, the worklist algorithm in [119] was adapted from a similar worklist algorithm in earlier work on process-modular analyses [153].

³Assuming no context sensitivity is added to the components.

- A choice of $\widehat{\text{State}}_{dep}$ to represent the part of the global analysis state associated with a particular dependency dep . For example, when dep is an address in the global store, $\widehat{\text{State}}_{dep}$ is used to represent the abstract value bound to dep in the global store. Put differently, the global analysis state is partitioned by dependencies, and $\widehat{\text{State}}_{dep}$ is used to represent that part for dependency dep . We require that each $\widehat{\text{State}}_{dep}$ forms a join-semilattice (Section 2.3.1) with bottom element $\perp_{\widehat{\text{State}}_{dep}}$ and join operator $(\sqcup_{\widehat{\text{State}}_{dep}})$ that adheres to the ACC (cf. condition 3 below).
- A function ANALYSE which is responsible for the analysis of a single component c , given the current analysis state S . This function returns a tuple (C, R, U) . C is the set of all components that were discovered during the analysis of component c (e.g., in a function-modular analysis, all functions that were called). R is the set of all dependencies that the analysis of component c depended on (e.g., some shared mutable state that was read). U is a (partial) map of updated dependencies; that is, it maps dependencies that were modified to the updated analysis state for that dependency (e.g., some shared mutable state and the value it was updated to). We require ANALYSE to be monotone, in the sense that if $S \sqsubseteq S'$, $(C, R, U) = \text{ANALYSE}(c, S)$ and $(C', R', U') = \text{ANALYSE}(c, S')$, then $C \subseteq C'$, $R \subseteq R'$ and $U \sqsubseteq U'$.

ALGORITHM 3: The sequential effect-driven worklist algorithm.

```

1   $W = \{c_0\}$ ,  $V = \{c_0\}$ ,  $S = \lambda dep. \perp_{\widehat{\text{State}}_{dep}}$ ,  $D = \lambda dep. \varnothing$ 
2  while  $W \neq \varnothing$  do
3      pick any  $c \in W$ 
4       $W := W \setminus \{c\}$ 
5       $(C, R, U) = \text{ANALYSE}(c, S)$ 
6       $W := W \cup (C \setminus V)$ 
7       $V := V \cup C$ 
8      foreach  $dep \in R$  do
9           $D := D[dep \mapsto D(dep) \cup \{c\}]$ 
10     foreach  $(dep, l) \in U$  do
11         if  $S(dep) \neq l$  then
12              $S := S[dep \mapsto l]$ 
13              $W := W \cup D(dep)$ 
    
```

Algorithm 3 maintains four iteration variables: a worklist W , a visited set V , the global analysis state S and a dependency map D . The worklist W contains all components that still need to be analysed (initially only containing c_0). The visited set V is used to keep track of all components that have already been

discovered during the analysis (initially also containing only c_0). The worklist algorithm repeatedly updates S , the global analysis state, until it converges to a fixpoint. The global analysis state is partitioned by dependencies. Therefore, it is represented using a (dependent) function of type $\Pi_{(dep:\widehat{\text{Dependency}})} \widehat{\text{State}}_{dep}$. The dependency map D tracks for each dependency dep the set of all components that depend on the analysis state associated with dep . This way, the algorithm can immediately determine which components need to be re-analysed when the analysis state associated with that dependency is updated.

The algorithm uses a sequential worklist iteration: as long as the worklist is not empty (line 2), it arbitrarily picks (line 3) and removes (line 4) a component c from W . This component is then analysed using the function `ANALYSE` (line 5). All previously unseen components are added to the worklist for analysis (line 6) and to the visited set V (line 7). For each dependency dep that was relied on during the analysis of component c , we add c to the set of components potentially impacted by a change to the analysis state associated with dep (line 9). For each dependency dep that was updated during the analysis of component c , we directly modify the analysis state associated with dep (line 12), and subsequently add all components potentially affected by this update to the worklist (line 13).

It can be shown that this worklist algorithm is always guaranteed to terminate as long as the following conditions are met:

1. The set of components $\widehat{\text{Component}}$ is chosen to be finite.
2. The set of dependencies $\widehat{\text{Dependency}}$ is chosen to be finite.
3. For each dependency dep , the analysis state associated with dep (i.e., the choice of $\widehat{\text{State}}_{dep}$) conforms to the ascending chain condition.
4. The component analysis function `ANALYSE` is monotone (as defined above).

Given these conditions, the order in which components are picked from the worklist (line 3) also does not influence the final result of the analysis (i.e., the analysis always converges to the same analysis state S). Nevertheless, the order in which components are analysed can influence in what sequence the analysis state converges and how many iterations are required in the algorithm. In practice, it can therefore have a significant impact on analysis performance.

Discussion. A key aspect of this effect-driven worklist algorithm is that it supports new components and dependencies being discovered during the analysis itself. It is exactly this aspect that renders this algorithm suitable for the analysis of dynamic, higher-order languages (such as JavaScript and Scheme), as in general the components and dependencies for a given program are not known upfront.

5.2.2 Instantiation for DSS

We now show how the generalised effect-driven worklist algorithm of Section 5.2.1 can be instantiated for a DSS-based analysis. As such, this worklist algorithm can be employed to efficiently compute the program semantics as the least fixpoint over the evaluation rules of DSS. First, we consider DSS with abstract GC, but without any form of store widening (as presented in Section 4.4). We consider each of the parameters that need to be provided for an instantiation of Algorithm 3.

Components. In this setting, components correspond to configurations that need to be evaluated (using the abstract big-step evaluation relation $\hat{\Downarrow}_{\hat{F}}^{\Delta}$). The initial component c_0 is the initial configuration for the program e under analysis.

$$\widehat{\text{Component}} = \widehat{\text{Config}} \qquad c_0 = \langle e, [], \perp_{\widehat{\text{Store}}}, \hat{t}_0 \rangle$$

Recall that, when using abstract counting and/or abstract GC with an infinite abstract domain (regardless of its adherence to the ACC), the analysis may fail to terminate due to a potentially infinite number of configurations. This would translate to a potentially infinite number of components (in violation of condition (1) for termination, as given in Section 5.2.1). Therefore, in order to guarantee termination, DSS-based analyses should either not use abstract GC or abstract counting, or instead switch to a finite abstract domain.

Dependencies. In DSS, the evaluation of one configuration using $\hat{\Downarrow}_{\hat{F}}^{\Delta}$ may require the result of the evaluation of another configuration (for recursive evaluation rules). The evaluation results for each configuration (that has been visited) are maintained in the global analysis state. Therefore, dependencies correspond to configurations for which the evaluation result is required.

$$\widehat{\text{Dependency}} = \widehat{\text{Config}}$$

Analysis State. For each dependency (i.e., configuration), the global analysis state keeps track of the evaluation results for the corresponding configuration. Due to the nondeterminism in the evaluation rules, a single configuration may have multiple evaluation results, which are kept together in a set.

$$\widehat{\text{State}}_{dep} = \mathcal{P}(\widehat{\text{Result}}) \quad (\forall dep \in \widehat{\text{Dependency}})$$

Given these definitions for dependencies and their associated analysis state, the global analysis state effectively corresponds to a *cache* of type $\widehat{\text{Cache}} = \widehat{\text{Config}} \rightarrow \mathcal{P}(\widehat{\text{Result}})$ that keeps track of the evaluation results for each encountered configuration. The effect-driven worklist algorithm given in Section 5.2.1 computes a fixpoint for this cache w.r.t. the mutually recursive evaluation relations $(\Downarrow^\Delta)$ and $(\Downarrow_F^\Delta)$ for the given program. The resulting cache S is equal to the cache computed by the iterative fixpoint computation of Darais et al. [40]. After termination, the final evaluation results for the top-level program expression e (i.e., the result of $\widehat{\text{eval}}(e)$) can be queried as $S(c_0)$.

Component Analysis. The analysis of a single component c (i.e., using the ANALYSE function) corresponds to the evaluation of configuration c . A key point is that during this evaluation, we can create new components for recursive evaluations of other configurations instead of evaluating these directly. We use the current cache to retrieve the latest version of the evaluation result for these configurations. Concretely, we adapt the definitions of $(\Downarrow^\Delta)$ and $(\Downarrow_F^\Delta)$ as follows:

- We pass along the current analysis state — i.e., a cache $\hat{X}$ — to reuse the analysis results for recursive evaluations. Instead of evaluating another configuration $\text{conf} \in \widehat{\text{Config}}$ directly, we can look up its current analysis result $\widehat{\text{res}} \in \widehat{\text{Result}}$ in the cache. However, we do not need to do this for every recursive evaluation: we only apply this caching mechanism at function call boundaries (i.e., in the E-APP rule), since this is the only place in these semantics where the evaluation could enter into a loop. Breaking this loop ensures that the computation of ANALYSE using $(\Downarrow_F^\Delta)$ becomes trivial. Another reason to use the cache only at function boundaries is that it increases the work that is done for the analysis of a single component using ANALYSE (because other configurations do still need to be evaluated). This in turn increases the potential for parallelisation later (cf. Section 5.3).
- The evaluation now also keeps track of the set of components $\hat{C}$ (i.e., configurations) for which the analysis results were retrieved from the cache, instead of being evaluated directly. This is necessary to ensure these components are in turn analysed (if they have not yet been discovered), and to register a dependency on these components for the analysis of the current component. The latter ensures that the current component gets re-analysed if the analysis results in the cache for any of these components are updated.

As a result of these changes, we now have that $(\Downarrow^\Delta), (\Downarrow_F^\Delta) \subseteq (\widehat{\text{Config}} \times \widehat{\text{Cache}}) \times (\widehat{\text{Result}} \times \mathcal{P}(\widehat{\text{Component}}))$. The updated evaluation rules are given below:

$$\begin{array}{c}
 \frac{\hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\langle ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \Downarrow^\Delta \langle \hat{v}, [], \emptyset, \emptyset \rangle, \emptyset} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\delta} = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\langle \text{set } x := ae, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \Downarrow^\Delta \langle \hat{v}, \hat{\delta}, \emptyset, \{\hat{a}\} \rangle, \emptyset} \quad [\text{E-SET}] \\
 \\
 \frac{\begin{array}{l} (\hat{f}, -) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad (\lambda x.e, \hat{\rho}') \in \hat{f} \\ \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}') \quad \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_x, \hat{v}_x) \\ \hat{\rho}'' = \hat{\rho}'[x \mapsto \hat{a}_x] \quad \hat{\sigma}' = \hat{\delta}_x \triangleright \hat{\sigma} \quad \hat{\sigma}'_{\text{gc}} = \hat{\sigma}'|_{\widehat{\mathcal{R}}_{(\hat{\sigma}')}}(\widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}'')) \end{array}}{\begin{array}{l} \widehat{\text{conf}} = \langle e, \hat{\rho}'', \hat{\sigma}'_{\text{gc}}, \hat{t}' \rangle \quad \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle \in \hat{X}(\widehat{\text{conf}}) \\ \langle f(ae), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \Downarrow^\Delta \langle \hat{v}, \hat{\delta} \triangleright \hat{\delta}_x, \{\hat{a}_x\} \cup \hat{A}, \hat{U} \rangle, \{\widehat{\text{conf}}\} \end{array}} \quad [\text{E-APP}] \\
 \\
 \frac{\begin{array}{l} (-, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (-, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\ \hat{v} = (\emptyset, \widehat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \end{array}}{\langle op(ae_1, ae_2), \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \Downarrow^\Delta \langle \hat{v}, [], \emptyset, \emptyset \rangle, \emptyset} \quad [\text{E-BINOP}] \\
 \\
 \frac{\begin{array}{l} \langle e_1, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \Downarrow_F^\Delta \langle \hat{v}_x, \hat{\delta}_1, \hat{A}_1, \hat{U}_1 \rangle, \hat{C}_1 \quad \hat{\delta}'_1 = \widehat{\text{replay}}(\hat{\delta}_1, \hat{\sigma}, \hat{A}_1) \\ \hat{\sigma}' = \hat{\delta}'_1 \triangleright \hat{\sigma} \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \quad \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x) \\ \langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \hat{\delta}_x \triangleright \hat{\sigma}', \hat{t} \rangle, \hat{X} \Downarrow_F^\Delta \langle \hat{v}, \hat{\delta}_2, \hat{A}_2, \hat{U}_2 \rangle, \hat{C}_2 \end{array}}{\begin{array}{l} \langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \\ \Downarrow^\Delta \langle \hat{v}, \hat{\delta}_2 \triangleright \hat{\delta}_x \triangleright \hat{\delta}'_1, \hat{A}_1 \cup \{\hat{a}_x\} \cup \hat{A}_2, \hat{U}_1 \cup \hat{U}_2 \rangle, \hat{C}_1 \cup \hat{C}_2 \end{array}} \quad [\text{E-LET}] \\
 \\
 \frac{\begin{array}{l} \hat{\sigma}_{\text{gc}} = \hat{\sigma}|_{\widehat{\mathcal{R}}_{(\hat{\sigma})}}(\widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho})) \\ \langle e, \hat{\rho}, \hat{\sigma}_{\text{gc}}, \hat{t} \rangle, \hat{X} \Downarrow^\Delta \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle, \hat{C} \\ \hat{\delta}_{\text{gc}} = \hat{\delta}|_{\widehat{\mathcal{R}}_{(\hat{\delta} \triangleright \hat{\sigma}_{\text{gc}})}}(\widehat{\mathcal{T}}_{\text{Val}}(\hat{v}) \cup \hat{U}_{\text{gc}}) \\ \hat{A}_{\text{gc}} = \hat{A} \cap \text{dom}(\hat{\delta}_{\text{gc}}) \quad \hat{U}_{\text{gc}} = \{\hat{a} \in \hat{U} \mid \hat{\sigma}_{\text{gc}}(\hat{a}) \neq \perp\} \end{array}}{\langle e, \hat{\rho}, \hat{\sigma}, \hat{t} \rangle, \hat{X} \Downarrow_F^\Delta \langle \hat{v}, \hat{\delta}_{\text{gc}}, \hat{A}_{\text{gc}}, \hat{U}_{\text{gc}} \rangle, \hat{C}} \quad [\text{E-GC}]
 \end{array}$$

Using these updated evaluation rules, we can define ANALYSE as follows:

$$\begin{aligned}
 \text{ANALYSE}(\widehat{\text{conf}}, \hat{X}) &= (C, R, U) \\
 \text{where } \hat{S} &= \{(\widehat{\text{res}}, \hat{C}) \mid \widehat{\text{conf}}, \hat{X} \Downarrow_{\hat{r}}^{\Delta} \widehat{\text{res}}, \hat{C}\} \\
 C = R &= \bigcup_{(-, \hat{C}) \in \hat{S}} \hat{C} \\
 U &= [\widehat{\text{conf}} \mapsto \{\widehat{\text{res}} \mid (\widehat{\text{res}}, -) \in \hat{S}\}]
 \end{aligned}$$

For a given component (i.e., configuration) $\widehat{\text{conf}}$, $\text{ANALYSE}(\widehat{\text{conf}}, \hat{X})$ computes the results for the evaluation of this configuration with respect to the current cache $\hat{X}$. Both C and R correspond to all configurations for which the results were retrieved from the analysis cache throughout this evaluation. The only part of the analysis state that is (potentially) modified as a result of this evaluation is the result for $\widehat{\text{conf}}$ in the cache. Therefore, U is a singleton set representing this update.

Note that, although the definition of ANALYSE is still formulated in terms of derivations of the evaluation relation $(\Downarrow_{\hat{r}}^{\Delta})$, these derivations are now trivially computable, as loops are no longer possible with these evaluation rules.⁴

5.2.3 Instantiation for DSS-FS

We now instantiate the generalised worklist algorithm for DSS-FS – i.e., for DSS with abstract GC *and* flow-sensitive store widening (as presented in Section 4.5). We revisit each of the parameters provided for an instantiation of Algorithm 3.

Components. The definitions of $\widehat{\text{Component}}$ and c_0 remain essentially unchanged with respect to the instantiation for DSS (cf. Section 5.2.2).

$$\widehat{\text{Component}} = \widehat{\text{Config}} \qquad c_0 = \langle e, [], \hat{t}_0 \rangle$$

Crucially, since configurations in DSS-FS no longer have their own separate store, we are guaranteed to always have a finite number of components. This is necessary to ensure the termination of Algorithm 3 (as described in Section 5.2.1).

⁴In fact, for λ_{ANF} , the only remaining rule with recursive evaluations is the E-LET rule. However, in a more conventional, richer language (e.g., one with if-expressions), evaluation rules with direct recursive evaluations would greatly outnumber those that instead retrieve the result from the cache (without any other looping constructs, this in fact happens only in E-APP).

Dependencies. For DSS-FS, we identify two different kinds of dependencies during the evaluation of configuration $\widehat{\text{conf}}$. As for DSS, the evaluation may depend on the evaluation result of other configurations. In addition, due to the flow-sensitive store widening, the evaluation requires access to the store that is associated with $\widehat{\text{conf}}$. Therefore, we define $\widehat{\text{Dependency}}$ as follows for DSS-FS:

$$\widehat{\text{Dependency}} ::= \text{result}(\widehat{\text{conf}}) \mid \text{store}(\widehat{\text{conf}})$$

Analysis State. Dependencies of the form $\text{result}(\widehat{\text{conf}})$ are associated with the evaluation result for $\widehat{\text{conf}}$. Unlike for DSS, the evaluation rules for DSS-FS are deterministic, so only a single evaluation result is kept per configuration. Dependencies of the form $\text{store}(\widehat{\text{conf}})$ are associated with the (widened) store for $\widehat{\text{conf}}$.

$$\widehat{\text{State}}_{\text{dep}} = \begin{cases} \widehat{\text{Result}} & \text{if } \text{dep} = \text{result}(\widehat{\text{conf}}) \\ \widehat{\text{Store}} & \text{if } \text{dep} = \text{store}(\widehat{\text{conf}}) \end{cases}$$

This leads to a global analysis state consisting of both (1) the cache $\hat{X}$ of evaluation results (as for DSS) and (2) a map $\hat{\Xi} : \widehat{\text{Config}} \rightarrow \widehat{\text{Store}}$, which maps each configuration to a single (widened) store. Under flow-sensitive store widening (cf. Section 4.5), all stores that can “flow to” a configuration $\widehat{\text{conf}}$ are joined together at $\hat{\Xi}(\widehat{\text{conf}})$. Similar to the cache of evaluation results $\hat{X}$, the store map $\hat{\Xi}$ is built up by the iterative fixpoint computation of Algorithm 3.

Component Analysis. We adapt the definitions for $(\hat{\Downarrow}^\Delta)$ and $(\hat{\Downarrow}_\Gamma^\Delta)$ as follows:

- We pass along the current cache $\hat{X}$ (as for DSS in Section 5.2.2).
- We also pass along the current store map $\hat{\Xi}$. Each rule uses the store $\hat{\Xi}(\widehat{\text{conf}})$ for the evaluation of configuration $\widehat{\text{conf}}$. Since $\hat{\Xi}$ only keeps track of the store for configurations which are also cached (i.e., in $\hat{X}$), we extend $\hat{\Xi}$ locally (e.g., in the E-LET rule) for direct (i.e., non-cached) recursive evaluations.
- The evaluation keeps track of a set $\hat{C}$ of all configurations for which cached analysis results were used (as for DSS), along with the store they were encountered with. The latter is necessary to correctly update $\hat{\Xi}$ for these configurations (i.e., by joining the encountered store into $\hat{\Xi}(\widehat{\text{conf}})$ for each $\widehat{\text{conf}}$). We represent $\hat{C}$ as a partial function of type $\widehat{\text{Config}} \rightarrow \widehat{\text{Store}}$, mapping each configuration in $\hat{C}$ to the store it was encountered with (more specifically, the join of all stores it was encountered with throughout the evaluation).

$$\begin{array}{c}
 \frac{\hat{\sigma} = \widehat{\Xi}(\widehat{\text{conf}}) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma})}{\underbrace{\langle ae, \hat{\rho}, \hat{t} \rangle, \hat{X}, \widehat{\Xi}}_{\widehat{\text{conf}}} \Downarrow^{\Delta} \langle \hat{v}, [], \emptyset, \emptyset \rangle, []} \quad [\text{E-ATOM}] \\
 \\
 \frac{\hat{\sigma} = \widehat{\Xi}(\widehat{\text{conf}}) \quad \hat{a} = \hat{\rho}(x) \quad \hat{v} = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \quad \hat{\delta} = \widehat{\text{update}}(\hat{\sigma}, \hat{a}, \hat{v})}{\underbrace{\langle \text{set } x := ae, \hat{\rho}, \hat{t} \rangle, \hat{X}, \widehat{\Xi}}_{\widehat{\text{conf}}} \Downarrow^{\Delta} \langle \hat{v}, \hat{\delta}, \emptyset, \{\hat{a}\} \rangle, []} \quad [\text{E-SET}] \\
 \\
 \begin{array}{l}
 \hat{\sigma} = \widehat{\Xi}(\widehat{\text{conf}}) \quad (\hat{f}, -) = \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad \hat{v}_x = \hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) \\
 (\lambda x_i. e_i, \hat{\rho}'_i) \in \hat{f} \quad \hat{t}' = \widehat{\text{tick}}(f, \hat{t}) \quad \hat{a}_i = \widehat{\text{alloc}}(x_i, \hat{t}') \\
 \hat{\delta}_i = \widehat{\text{extend}}(\hat{\sigma}, \hat{a}_i, \hat{v}_x) \quad \hat{\rho}''_i = \hat{\rho}'_i[x_i \mapsto \hat{a}_i] \quad \hat{\sigma}_i = \hat{\delta}_i \triangleright \hat{\sigma} \\
 \widehat{\text{conf}}_i = \langle e_i, \hat{\rho}''_i, \hat{t}' \rangle \quad \hat{\sigma}_{i_{\text{gc}}} = \hat{\sigma}'_i |_{\widehat{\mathcal{R}}_{(\sim_{\hat{\sigma}'_i})}(\widehat{\mathcal{T}}_{\text{Env}}(\hat{\rho}''_i))} \\
 \langle \hat{v}_i, \hat{\delta}'_i, \hat{A}_i, \hat{U}_i \rangle = \hat{X}(\widehat{\text{conf}}_i)
 \end{array} \\
 \frac{\hat{v} = \bigsqcup_i \hat{v}_i \quad \hat{\delta} = \bigsqcup_{\hat{\sigma}} \hat{\delta}'_i \triangleright \hat{\delta}_i \quad \hat{A} = \bigcup_i \{\hat{a}_i\} \cup \hat{A}_i \quad \hat{U} = \bigcup_i \hat{U}_i}{\underbrace{\langle f(ae), \hat{\rho}, \hat{t} \rangle, \hat{X}, \widehat{\Xi}}_{\widehat{\text{conf}}} \Downarrow^{\Delta} \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle, \bigsqcup_i [\widehat{\text{conf}}_i \mapsto \hat{\sigma}_{i_{\text{gc}}}] } \quad [\text{E-APP}] \\
 \\
 \hat{\sigma} = \widehat{\Xi}(\widehat{\text{conf}}) \\
 \frac{\begin{array}{l}
 \widehat{\text{conf}}_1 \\
 \underbrace{\langle e_1, \hat{\rho}, \hat{t} \rangle, \hat{X}, \widehat{\Xi}[\widehat{\text{conf}}_1 \mapsto \hat{\sigma}]}_{\widehat{\text{conf}}_1} \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}_x, \hat{\delta}_1, \hat{A}_1, \hat{U}_1 \rangle, \hat{C}_1 \\
 \hat{\delta}'_1 = \widehat{\text{replay}}(\hat{\delta}_1, \hat{\sigma}, \hat{A}_1) \quad \hat{\sigma}' = \hat{\delta}'_1 \triangleright \hat{\sigma} \quad \hat{a}_x = \widehat{\text{alloc}}(x, \hat{t}) \\
 \hat{\delta}_x = \widehat{\text{extend}}(\hat{\sigma}', \hat{a}_x, \hat{v}_x) \quad \hat{\sigma}'' = \hat{\delta}_x \triangleright \hat{\sigma}' \\
 \underbrace{\langle e_2, \hat{\rho}[x \mapsto \hat{a}_x], \hat{t} \rangle, \hat{X}, \widehat{\Xi}[\widehat{\text{conf}}_2 \mapsto \hat{\sigma}'']}_{\widehat{\text{conf}}_2} \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_2, \hat{A}_2, \hat{U}_2 \rangle, \hat{C}_2
 \end{array}}{\underbrace{\langle \text{let } x = e_1 \text{ in } e_2, \hat{\rho}, \hat{t} \rangle, \hat{X}, \widehat{\Xi}}_{\widehat{\text{conf}}} \Downarrow^{\Delta} \langle \hat{v}, \hat{\delta}_2 \triangleright \hat{\delta}_x \triangleright \hat{\delta}'_1, \hat{A}_1 \cup \{\hat{a}_x\} \cup \hat{A}_2, \hat{U}_1 \cup \hat{U}_2 \rangle, \hat{C}_1 \sqcup \hat{C}_2} \quad [\text{E-LET}]
 \end{array}$$

$$\begin{array}{c}
 \hat{\sigma} = \hat{\Xi}(\widehat{\text{conf}}) \quad (-, \hat{n}_1) = \hat{\mathcal{A}}(ae_1, \hat{\rho}, \hat{\sigma}) \quad (-, \hat{n}_2) = \hat{\mathcal{A}}(ae_2, \hat{\rho}, \hat{\sigma}) \\
 \hat{v} = (\emptyset, \hat{\mathcal{I}}(op, \hat{n}_1, \hat{n}_2)) \\
 \hline
 \underbrace{\langle op(ae_1, ae_2), \hat{\rho}, \hat{t} \rangle, \hat{X}, \hat{\Xi}}_{\widehat{\text{conf}}} \Downarrow^{\Delta} \langle \hat{v}, [], \emptyset, \emptyset \rangle, [] \quad [\text{E-BINOP}] \\
 \\
 \hat{\sigma}_{gc} = \hat{\Xi}(\widehat{\text{conf}}) \quad \langle e, \hat{\rho}, \hat{t} \rangle, \hat{X}, \hat{\Xi} \Downarrow^{\Delta} \langle \hat{v}, \hat{\delta}, \hat{A}, \hat{U} \rangle, \hat{C} \\
 \hat{\delta}_{gc} = \hat{\delta}|_{\hat{\mathcal{R}}_{(\hat{\sim}_{\hat{\delta} \triangleright \hat{\sigma}_{gc}})}(\hat{\mathcal{T}}_{\text{Val}}(\hat{v}) \cup \hat{U}_{gc})} \\
 \hat{A}_{gc} = \hat{A} \cap \text{dom}(\hat{\delta}_{gc}) \quad \hat{U}_{gc} = \{\hat{a} \in \hat{U} \mid \hat{\sigma}_{gc}(\hat{a}) \neq \perp\} \\
 \hline
 \underbrace{\langle e, \hat{\rho}, \hat{t} \rangle, \hat{X}, \hat{\Xi}}_{\widehat{\text{conf}}} \Downarrow_{\hat{\Gamma}}^{\Delta} \langle \hat{v}, \hat{\delta}_{gc}, \hat{A}_{gc}, \hat{U}_{gc} \rangle, \hat{C} \quad [\text{E-GC}]
 \end{array}$$

Using these updated evaluation rules, we can define ANALYSE as follows:

$$\begin{array}{l}
 \text{ANALYSE}(\widehat{\text{conf}}, S) = (C, R, U) \\
 \text{where } \hat{X}(\widehat{\text{conf}}) = S(\text{result}(\widehat{\text{conf}})) \\
 \hat{\Xi}(\widehat{\text{conf}}) = S(\text{store}(\widehat{\text{conf}})) \\
 \widehat{\text{conf}}, \hat{X}, \hat{\Xi} \Downarrow_{\hat{\Gamma}}^{\Delta} \widehat{\text{res}}, \hat{C} \\
 C = \{\widehat{\text{conf}}' \mid \widehat{\text{conf}}' \in \text{dom}(\hat{C})\} \\
 R = \{\text{result}(\widehat{\text{conf}}') \mid \widehat{\text{conf}}' \in C\} \cup \{\text{store}(\widehat{\text{conf}})\} \\
 U = [\text{result}(\widehat{\text{conf}}) \mapsto \widehat{\text{res}}] \sqcup \\
 \bigsqcup_{\widehat{\text{conf}}' \in \text{dom}(\hat{C})} [\text{store}(\widehat{\text{conf}}') \mapsto \hat{\Xi}(\widehat{\text{conf}}') \sqcup \hat{C}(\widehat{\text{conf}}')]
 \end{array}$$

Function ANALYSE first splits the given analysis state S into a cache $\hat{X}$ and store map $\hat{\Xi}$, which are used to evaluate the given configuration $\widehat{\text{conf}}$. The set of created components C is directly available from the evaluation. The set of read dependencies R contains both dependencies on the evaluation results of other configurations, as well as a dependency on the store associated with the current configuration (the only cached configuration for which the store is read from the analysis state). The map of updated dependencies U contains the updated result for the analysed component (as for DSS) plus, for each configuration $\widehat{\text{conf}}'$ in C , the store that $\widehat{\text{conf}}'$ was encountered with. Specifically, all stores that are known to flow to $\widehat{\text{conf}}'$ are joined together with the existing store at $\hat{\Xi}(\widehat{\text{conf}}')$.

5.3 A Parallel Worklist Algorithm for DSS

Algorithm 3 shows how an analysis fixpoint can be computed efficiently using a sequential, effect-driven worklist algorithm. In this section, we propose a novel *parallel* version of this worklist algorithm, obtaining the exact same result as its sequential counterpart. A key benefit of parallelising this generalised worklist algorithm is that its existing instantiations (such as those for DSS and DSS-FS) are automatically rendered parallel for free (i.e., without any modifications).

We first present our core parallel worklist algorithm (Section 5.3.1). Then, we propose some optimisations that can be applied to this algorithm to further increase its parallel efficiency (Section 5.3.2).

5.3.1 Parallel Effect-Driven Worklist Algorithm

The core idea for a parallel variant of the effect-driven worklist algorithm (Algorithm 3) is to exploit the inherent opportunity for worklist-based parallelism. That is, multiple components in the worklist W can be analysed in parallel. A key benefit of the modular analysis design is that it automatically provides coarse-grained tasks (i.e., analyses of components) that can be run in parallel because they are already computed in isolation from one another.

More specifically, our parallel worklist algorithm uses a separate *worker* thread for each component in the worklist that needs to be analysed. To avoid synchronisation costs in updating the global analysis state (i.e., S in Algorithm 3), each component is analysed using its own local copy of that analysis state. A single *coordinator* thread is responsible for processing the incoming results of the component analyses and subsequently updating the global analysis state. Our parallel worklist algorithm thus follows the “coordinator-worker” paradigm, which can easily be implemented using several concurrency mechanisms (e.g., using actors). It is important to highlight that processing results using a single coordinator thread introduces a sequential bottleneck, but trivially avoids issues with race conditions. That is, we are theoretically limited by the rate at which the coordinator can “commit” the results for the analysis of a component to the global analysis state. We expect throughput to be mostly dominated by the cost of the component analyses, which can be processed in parallel using multiple workers.

There are two key insights that we leverage for the parallel worklist algorithm:

- When multiple component analyses are executed concurrently, it may occur that one updates some part of the global analysis state that the other depends on. Since the analysis state is kept local to each worker, such updates

happening in one component analysis are not visible in another.⁵ Therefore, after each component analysis, it should be checked that no part of the analysis state that was read during that component analysis has been updated in the meantime; if so, the component should be analysed again. We do not require any additional bookkeeping for this, as we can exploit the dependencies that the component analysis relied on (i.e., the set R returned by the ANALYSE function) to check if another (parallel) component analysis might have interfered in its computation.

- We can exploit the monotonicity of the analysis when updating the analysis state. If a component analysis updates some part of the analysis state, we can always apply these updates to the global analysis state, *even if* the local analysis state used during the component analysis is no longer up-to-date (i.e., not using the latest S). The reason is the following: this local analysis state would always be subsumed by any newer state (because S only “grows”), so that by the monotonicity of the component analysis (cf. the monotonicity assumption for ANALYSE), we can guarantee that an updated result computed using this newer state would subsume any previously computed result. Because of the associative and commutative properties of the join operator, we can always apply the computed updates to the current analysis state, even if that analysis state is more up-to-date than the local analysis state that they were computed with. Again, we do not require any additional bookkeeping for this, as we can exploit the dependencies that the component analysis updated (i.e., the set U returned by the ANALYSE function) to track all updates to the analysis state after a component analysis.

Note that, the resulting approach to parallelism has certain similarities to STM: when analysis state that was read by one component analysis (analogous to mutable state read by an STM transaction) is updated by another parallel component analysis (i.e., written to by another parallel STM transaction), the component analysis needs to be re-executed (i.e., the STM transaction is retried). However, unlike in traditional STM, we do not need to throw away the results of a component analysis (i.e., the updates performed by an STM transaction) that needs to be retried. Instead, by exploiting the monotonicity of the analysis, we can still integrate those changes to the analysis state, speeding up its convergence.

The parallel worklist algorithm is shown below in Algorithm 4.

⁵Even if the changes computed during the component analysis were directly performed on some shared, mutable analysis state instead, one component analysis may have already read part of this state before another parallel component analysis updated that part of the state.

ALGORITHM 4: The parallel effect-driven worklist algorithm.

```

1   $P = \{c_0\}, V = \{c_0\}, S = \lambda dep. \perp_{\widehat{\text{State}_{dep}}}, D = \lambda dep. \emptyset$ 
2  function SCHEDULE( $c$ ):
3      fork
4           $S_{local} = S$ 
5           $result = \text{ANALYSE}(c, S_{local})$ 
6          send ( $c, S_{local}, result$ ) to coordinator
7  SCHEDULE( $c_0$ )
8  while  $P \neq \emptyset$  do
9      wait for next result ( $c, S_{local}, (C, R, U)$ )
10     foreach  $c' \in C$  do
11         if  $c' \notin V$  then
12              $V := V \cup \{c'\}$ 
13              $P := P \cup \{c'\}$ 
14             SCHEDULE( $c'$ )
15     foreach  $dep \in R$  do
16          $D := D[dep \mapsto D(dep) \cup \{c\}]$ 
17     foreach  $(dep, l) \in U$  do
18         if  $l \not\sqsubseteq S(dep)$  then
19              $S := S[dep \mapsto S(dep) \sqcup l]$ 
20             foreach  $c' \in D(dep)$  do
21                 if  $c' \notin P$  then
22                      $P := P \cup \{c'\}$ 
23                     SCHEDULE( $c'$ )
24     if  $\exists dep \in R : S_{local}(dep) \neq S(dep)$  then
25         SCHEDULE( $c$ )
26     else
27          $P := P \setminus \{c\}$ 
    
```

Similar to the sequential algorithm, it keeps track of a visited set V , the global analysis state S and a dependency map D (line 1). However, there is no longer a worklist W : components that need to be analysed are directly sent off to a worker. This happens using the function SCHEDULE (lines 2-6): a new task is forked (which we assume is eventually assigned to a worker thread) that will analyse the given component (using the ANALYSE function) and send back the result to the

coordinator when finished.⁶ A set P keeps track of which components are currently scheduled (in parallel) by this function, ensuring that a single component is never scheduled multiple times at any given moment.

To kickstart the analysis, we schedule the initial component c_0 (line 7). The coordinator thread then enters a loop (lines 8-27) which stops once all the scheduled components have been analysed (i.e., when the set P is empty). In each iteration, it updates the analysis state after receiving an incoming result of a component analysis (computed by a worker) of some component c (line 9).

Analogous to the sequential algorithm, it first schedules every discovered component in C that has not yet been visited⁷ (lines 10-14), then registers the read dependencies R of the component that was analysed (lines 15-16).

Updating the analysis state becomes more complicated for the parallel algorithm, as we have to take into account that S has been updated during the analysis of component c (in which case $S_{local} \neq S$). For an updated dependency $(dep, l) \in U$, we are guaranteed (cf. assumptions on function ANALYSE) that $S_{local}(dep) \sqsubseteq l$, but not that $S(dep) \sqsubseteq l$. As such, we cannot just replace $S(dep)$ with l ; instead, we need to apply each update to the current analysis state using a join. Concretely, we first check (line 18) for every updated dependency dep , if the updated value l is already subsumed by the current value in the analysis state $S(dep)$. If so, this means that another (parallel) component analysis already updated $S(dep)$ in a way that the current update is already incorporated in the analysis state. Otherwise, we join the current value $S(dep)$ with the updated value l (line 19). Again, we cannot just replace $S(dep)$ with l (as we did in the sequential algorithm), since $S(dep)$ may have been updated so that $S(dep) \not\sqsubseteq l$ and $l \not\sqsubseteq S(dep)$. The monotonicity of the analysis combined with the commutative property of the join operator allows us to join both values, as the order in which updates to the analysis state happen does not influence the outcome. By performing updates to the analysis state in this way, the parallel analysis can speed up its convergence. As in the sequential algorithm, we also schedule all components that may have been affected by the triggered dependency dep (lines 20-23).

Finally, we must check if component c needs to be re-analysed because some part of the analysis state it depends on (as indicated by R) was updated. Concretely, we check if any dependency $dep \in R$ that was used during the component analysis has been updated in the meantime (line 24). If so, we must re-analyse component c to take into account this updated analysis state (line 25). If not, the analysis result of c has been processed, meaning that we can remove it from P (line 27).

⁶In an actual implementation, it is necessary to make S a `volatile` variable. The coordinator thread modifies S , while worker threads only read S . It is not necessary for a worker to have the latest version of S ; variable S_{local} is introduced to ensure a single version of S is read.

⁷Note that $c' \notin V$ implies $c' \notin P$ here.

Discussion. The parallelism that can be exploited by this algorithm will depend on the number of components that are scheduled at the same time (i.e., based on the size of the set P). The analysis behaviour is non-deterministic in the order in which scheduled components are analysed, the order in which concurrent component analyses finish, and the order in which their results are processed. This significantly influences in what order and in how many iterations the analysis state will converge; however, it has no impact on the final analysis result, which regardless of this order is always the same as that of Algorithm 3.

5.3.2 Optimising for Parallel Efficiency

We present three optimisations that can be applied to Algorithm 4 to further enhance its parallel efficiency. These are all aimed at reducing the workload of the single-threaded coordinator, so that it can process analysis results from the workers faster. This allows for greater speedups, as the sequential bottleneck is reduced. We present the optimised parallel worklist algorithm in Algorithm 5, and discuss each of the optimisations that have been integrated.

Avoiding Joins for Private Dependencies. Recall that the parallel worklist algorithm — unlike its sequential counterpart — needs to update dependencies by joining the result of the component analysis l for a dependency dep with the current state $S(dep)$ (line 25). The reason for this is that a dependency can be updated by multiple component analyses running in parallel, which requires all contributions to be joined. Some dependencies, however, can only be updated by the analysis for a single component (such as the dependency on the result of the component being analysed). We refer to such dependencies as *private dependencies*. Since our algorithm guarantees that there can never be multiple parallel analyses of the same component, the coordinator does not need to perform a join operation with the existing analysis state for these private dependencies.

Concretely, we assume each instantiation provides a predicate `IsPrivate`, which determines which dependencies can be marked as private. For DSS, the only dependencies are those related to the result of a component, which are only updated by the analysis of said component. Therefore, all dependencies for DSS can be considered private. For DSS-FS, the same dependencies on component results are also private, whereas dependencies on the store used for a component are not (since these can be updated by any “caller” component). Subsequently, the parallel algorithm is modified to directly update the analysis state (i.e., skipping the join with the existing analysis state) for updated dependencies that satisfy the `IsPrivate` predicate (line 23). For larger analysis states, this can meaningfully reduce the workload for the single-threaded coordinator.

ALGORITHM 5: The optimised parallel worklist algorithm.

```

1   $P = \{c_0\}, V = \{c_0\}, S = \lambda dep. \perp_{\widehat{\text{State}_{dep}}}, D = \lambda dep. \emptyset, T = \lambda dep. 0$ 
2  function SCHEDULE( $c$ ):
3      fork
4           $(S_{local}, T_{local}) = (S, T)$ 
5           $(C, R, U) = \text{ANALYSE}(c, S_{local})$ 
6           $C' = C \setminus V$ 
7           $R' = R \setminus \{dep \in R \mid c \in D(dep)\}$ 
8           $U' = \{(dep, \text{DIFF}(l, S(dep))) \mid (dep, l) \in U \wedge l \neq S(dep)\}$ 
9          send  $(c, S_{local}, R, T_{local}, (C', R', U'))$  to coordinator
10 SCHEDULE( $c_0$ )
11 while  $P \neq \emptyset$  do
12     wait for next result  $(c, S_{local}, R_{all}, T_{local}, (C, R, U))$ 
13     foreach  $c' \in C$  do
14         if  $c' \notin V$  then
15              $V := V \cup \{c'\}$ 
16              $P := P \cup \{c'\}$ 
17             SCHEDULE( $c'$ )
18     foreach  $dep \in R$  do
19          $D := D[dep \mapsto D(dep) \cup \{c\}]$ 
20     foreach  $(dep, l) \in U$  do
21         if  $l \not\sqsubseteq S(dep)$  then
22             if ISPRIVATE( $dep$ ) then
23                  $S := S[dep \mapsto l]$ 
24             else
25                  $S := S[dep \mapsto S(dep) \sqcup l]$ 
26                  $T := T[dep \mapsto T(dep) + 1]$ 
27             foreach  $c' \in D(dep)$  do
28                 if  $c' \notin P$  then
29                      $P := P \cup \{c'\}$ 
30                     SCHEDULE( $c'$ )
31     if  $\exists dep \in R_{all} : T_{local}(dep) \neq T(dep)$  then
32         SCHEDULE( $c$ )
33     else
34          $P := P \setminus \{c\}$ 

```

Filtering Component Analysis Results. The cost of processing the results is directly linked to the size of the C , R and U sets returned by the ANALYSE function. We can reduce the workload on the single-threaded coordinator by first filtering these sets in the worker (preserving only the results that can be considered “new”) after computing the component analysis. Specifically, the worker can remove all components in C which are already included in the visited set V , since these are discarded anyway on line 11 in Algorithm 4. Similarly, it can filter the sets R and U to only include dependencies that are not yet registered for the analysed component or where the corresponding updates are not yet subsumed by the current analysis state, respectively. Note that even if the worker pre-filters its analysis results, the coordinator cannot leave out those same filters: the analysis state may have been updated after the worker’s filtering of the results, but before they are processed by the coordinator. On the other hand, the filtering itself is always safe due to the increasing nature of the analysis (e.g., the set V only grows, so the filtering may only have been too conservative).

To incorporate this optimisation, we extend the worker’s thread spawned in the SCHEDULE function in Algorithm 5. After computing the component analysis result (C, R, U) (using function ANALYSE), the worker now filters these results into an optimised result (C', R', U') . The set C is filtered to only contain components that are not yet in the visited set V (line 6), since components that are already in both C and V are simply skipped by the coordinator (line 14). The set R is filtered to only contain dependencies that are new for the analysed component (line 7), since existing dependencies do not actually contribute to D (line 19). The map U is filtered to only contain the difference between the updated dependencies and the current global analysis state (line 8). That is, we remove each updated dependency (dep, l) where $S(dep) = l$, as these would otherwise be skipped by the coordinator (line 21). For the updated dependencies that remain, we only preserve the part of the analysis state that is not yet subsumed by the current analysis state. For this purpose, we employ a function $\text{DIFF} : \widehat{\text{State}}_{dep} \times \widehat{\text{State}}_{dep} \rightarrow \widehat{\text{State}}_{dep}$, which takes the updated analysis state (associated with the updated dependency) and the current analysis state, and computes the “difference” between both. Since this function is dependent on the representation of $\widehat{\text{State}}_{dep}$, it is instantiation-specific. For the instantiation of DSS, there is no need to make use of DIFF (i.e., we can define DIFF as the identity function), since all its dependencies are private. For the instantiations of DSS-FS, we define function DIFF as follows:

$$\text{DIFF}(l, l_{prv}) = \begin{cases} l & \text{if } l, l_{prv} \in \widehat{\text{Result}} \\ l|_{\{a \mid l(a) \neq l_{prv}(a)\}} & \text{if } l, l_{prv} \in \widehat{\text{Store}} \end{cases}$$

Note that the filtering of U happens with respect to the current global analysis state S , and not the state S_{local} used for the analysis of the component. Doing so ensures that potentially more dependencies can be filtered out (since $S_{local} \sqsubseteq S$), thereby further reducing the workload for the single-threaded coordinator.

Timestamped Dependencies. In practice, checking if no read dependencies were changed after a component analysis by using equality checks on the corresponding parts of the analysis state (i.e., line 24 in Algorithm 4) could be somewhat expensive, at least in this context where we want to avoid a sequential bottleneck by reducing the workload of the coordinator. A simple workaround is to assign a *timestamp* to each dependency (initially 0 for every dependency). Whenever the analysis state is updated, we increase the timestamp of the corresponding dependency. As such, we can eliminate the expensive equality checks on analysis state, instead replacing them with much cheaper equality checks on the timestamps of the dependencies. Using such timestamps for cheaper comparisons goes back to earlier work on optimising the performance of static analysers [147, 58].

In Algorithm 5, we integrate these timestamps by maintaining an additional data structure T , which maps each dependency to a timestamp (initially 0). Each update to the analysis state associated with a dependency increases the timestamp of that dependency (line 26). The increased parallel efficiency comes from the cheaper comparisons of timestamps (instead of analysis state) in the coordinator thread (line 31). Note that, on line 4, it is important that the pair (S_{local}, T_{local}) is initialised atomically, so that the local timestamps match the local analysis state.

5.4 Evaluation

The sequential worklist algorithm of Section 5.2 is used by default in the MAF framework [165] to drive modular analyses. We have implemented the instantiations for DSS (Section 5.2.2) and DSS-FS (Section 5.2.3) in this framework (i.e., expressed as modular analyses). In our experiments, we can therefore measure the efficiency improvement of the effect-driven worklist algorithm (compared to a naive Kleene iteration) to compute the fixpoint for these DSS-based analyses.

Our parallel worklist algorithm (Section 5.3.1), including the optimisations discussed in Section 5.3.2, has also been integrated into the MAF framework, where it is made available as a Scala trait that can directly be used with any existing modular analysis. We set up our experiments (Section 5.4.1) using this implementation, and measure the speedups of the parallel version for an increasing number of threads compared to the sequential worklist algorithm of Section 5.2.

In addition, we have also verified the correctness of the parallel implementations by running the built-in soundness tests of MAF and by verifying that analysis results are the same as those of the sequential implementation for all benchmarks.

Specifically, we aim to answer the following research questions:

- RQ1** How efficient is the effect-driven worklist algorithm to compute DSS-based analyses compared to a naive fixpoint iteration?
- RQ2** What speedups can the parallel worklist algorithm achieve for DSS?
- RQ3** What speedups can the parallel worklist algorithm achieve for other instantiations of the worklist algorithm such as MODF?

5.4.1 Experimental Setup

We describe the setup of our experiments. For RQ1, we use a slightly different setup, as these experiments only pertain to the sequential worklist algorithm, and therefore do not need to be executed in a multi-core environment. For RQ2 and RQ3, we run experiments on different machines featuring a high number of CPU cores, and employ a separate benchmark suite to ensure that benchmarks take long enough to increase the potential for meaningful speedups.

Environment. We ran all experiments for RQ1 using Java version 17 (and Scala version 3.1.0) on an Apple Mac Mini with an M2 Pro CPU and 16GB of RAM. For both RQ2 and RQ3, we make use of different machines with a higher number of CPU cores, running all experiments on an HPC cluster. This cluster runs each benchmark on either a (shared) node with a 2×32 -core AMD EPYC 9384X CPU, or a 2×64 -core AMD EPYC 9535 CPU.⁸ All machines in the HPC cluster use Java version 25, and we configured the JVM with a maximum heap size of 64GB.

Implementation. We implemented both the sequential (Section 5.2) and parallel (Section 5.3) worklist algorithms into the MAF framework. We also provided instantiations (which are shared for both the sequential and parallel algorithm) for both DSS (Section 4.4) and DSS-FS (Section 4.5). In order to avoid measuring mostly the overhead of abstract (tracing) GC, we modified the implementations of DSS and DSS-FS to no longer apply abstract GC at every step, but only at every function call. For RQ3, we configured the existing instantiation of MODF in MAF with our parallel worklist algorithm.

⁸While these machines feature different performance characteristics, RQ2 and RQ3 only measure relative speedups, and all measurements of the same benchmark run on the same machine.

Benchmarks. For RQ1, we reuse the same benchmarks as for the evaluation in Section 4.6, and refer to Table 4.1 for an overview of their characteristics. For RQ2 and RQ3, we keep the same benchmarks for DSS. However, for DSS-FS and MODF, these benchmarks are not adequate: as we measure speedups for configurations up to 32 workers, we require that the sequential benchmark takes long enough so that it offers opportunities for meaningful speedups. Therefore, we use an additional benchmark suite provided by MAF, for which the characteristics (LOC and sequential running time) are summarised in Table 5.1. When reporting results for RQ2 and RQ3, for each configuration we filter out benchmark programs for which the corresponding sequential benchmark took less than 1 second, or where the sequential implementation timed out (using a timeout of 2 hours).

Benchmark	LOC	DSS-FS ($m = 0$)	DSS-FS ($m = 2$)	MODF ($m = 0$)	MODF ($m = 2$)
multiple-dwelling	402	14m38s	∞	1s	∞
decompose	412	25m39s	∞	2s	∞
prime-sum-pair	394	9m13s	∞	1s	∞
eceval	774	∞	∞	43s	4m7s
peval	497	32m29s	∞	3s	∞
scheme	767	∞	∞	5s	∞
sboyer	632	1m9s	2m28s	11s	21s
matrix	617	2s	31s	74ms	2s
nboyer	625	1m5s	2m42s	5s	16s
browse	161	1s	2m20s	86ms	2s
compiler	466	26s	13m28s	417ms	23s
leval	334	49s	5m33s	343ms	5s

Table 5.1: An overview of the extra benchmarks for RQ2 and RQ3, along with their LOC (as calculated by `clloc`) and running times for the sequential implementation of DSS-FS and MODF (each with $m = 0$ and $m = 2$). Benchmarks that timed out (after 2 hours) are marked using ∞ .

For all experiments, we use m -CFA [109] for context sensitivity (which uses the top m stack frames as context), where higher values of m may increase precision. For the abstract domain, we employ a constant propagation lattice (Section 2.3.1) as an abstraction for primitive domains in all experiments.

Measurements. Each experiment was preceded by 10 warm-up runs (with a total timeout of 1 minute), and reported results are the median of 3 runs. As advocated by Dewey et al. [42], we report speedups of the parallel worklist algorithm relative to the sequential implementation (which is not always the case for speedups reported in other related work [93, 101, 114, 175]).

5.4.2 RQ1: Impact of the Effect-Driven Worklist Algorithm for DSS

The effect-driven worklist algorithm of Section 5.2, along with the instantiations of Section 5.2.2 and Section 5.2.3, offer a more efficient fixpoint computation for DSS and DSS-FS compared to a naive fixpoint computation (i.e., Kleene iteration). Such a naive fixpoint computation analyses all components using the current analysis state in each iteration, and terminates once the analysis state has converged (i.e., when none of the component analyses further modify the analysis state). In contrast, the effect-driven worklist algorithm of Section 5.2 is a more directed chaotic fixpoint iteration, which in each iteration only analyses a single component that may have been affected by a change (i.e., effect) to the analysis state in a previous iteration (by keeping track of dependencies).

To quantify the efficiency improvements brought by the effect-driven worklist algorithm for DSS and DSS-FS, we compare their instantiations (Sections 5.2.2 and 5.2.3) with an implementation of DSS using a naive fixpoint algorithm (specifically, the Kleene iteration described in the previous paragraph). We compare both the total time taken for the analysis, as well as the total number of iterations (i.e., the total number of component analyses with the ANALYSE function). The results are shown in Table 5.2 (for DSS⁹) and Table 5.3 (for DSS-FS).

	time (ms)				#iterations		
	naive	effect-driven	%improvement		naive	effect-driven	%improvement
cpstak	17	11	35.29%		137	45	67.15%
dderiv	138 273	4 281	96.90%		110 184	2 777	97.48%
deriv	5 441	185	96.60%		12 873	318	97.53%
diviter	19 094	367	98.08%		85 911	731	99.15%
divrec	20 528	338	98.35%		84 993	725	99.15%
puzzle	>1 801 232	315 554	>82.48%	>13 610 477	140 251	>98.97%	
takl	412	49	88.11%		1 682	128	92.39%
triangl	396 538	5 391	98.64%		554 031	9 934	98.21%
tak	8 553	100	98.83%		62 984	567	99.10%
regex	323	18	94.43%		3 114	121	96.11%
rsa	187	47	74.87%		1 421	127	91.06%

Table 5.2: Comparison of time taken and total number of iterations between a naive and effect-driven fixpoint algorithm for DSS.

Both DSS and DSS-FS significantly benefit using an effect-driven worklist algorithm, compared to a naive iteration strategy. The improvements are more pronounced for DSS than for DSS-FS. The reason for this is that an analysis using DSS (without any form of store widening) encounters many different components (as the store is part of each component). The naive fixpoint algorithm

⁹For DSS, we have omitted benchmarks that timed out for both iteration strategies.

	time (ms)			#iterations		
	naive	effect-driven	%improvement	naive	effect-driven	%improvement
boyer	174 017	113 013	35.06%	2 440	1 944	20.33%
browse	2 418	953	60.59%	1 484	558	62.40%
cpstak	3	3	0.00%	45	30	33.33%
dderiv	392	164	58.16%	503	225	55.27%
deriv	207	87	57.97%	242	109	54.96%
destruc	339	83	75.52%	806	152	81.14%
diviter	1 026	122	88.11%	751	124	83.49%
divrec	1 092	150	86.26%	642	120	81.31%
puzzle	113	51	54.87%	434	160	63.13%
takl	26	13	50.00%	101	37	63.37%
triangl	73	27	63.01%	175	57	67.43%
matrix	2 577	1 784	30.77%	5 358	2 565	52.13%
tak	2	2	0.00%	27	13	51.85%
mceval	4 083	2 950	27.75%	3 624	2 397	33.86%
regex	160	119	25.62%	546	340	37.73%
rsa	27	9	66.67%	214	54	74.77%

Table 5.3: Comparison of time taken and total number of iterations between a naive and effect-driven fixpoint algorithm for DSS-FS.

needs to analyse all of these components in each iteration. In contrast, the effect-driven worklist algorithm selectively analyses only those components that could potentially be affected by an update to the analysis state. Furthermore, due to the store widening in DSS-FS, component analyses have more dependencies (whereas in DSS, they only have dependencies on the results of their callees). This renders components more “sensitive” to updates in the global analysis state (whereas in DSS, components only require re-analysis when the results of their callees change).

5.4.3 RQ2: Parallel Efficiency for DSS

Figure 5.1 shows the speedups for both context-insensitive (i.e., using $m = 0$) and context-sensitive (i.e., using $m = 2$) DSS and DSS-FS when using our parallel worklist algorithm. For DSS, a total of only 6 benchmarks from the original benchmarking suite met our inclusion criteria (as described in Section 5.4.1).

For most configurations and benchmarks, we observe rather consistent speedups up to 16 workers. For DSS, we achieve a maximum speedup of $41\times$ for the context-insensitive configuration with 32 workers (on the **puzzle** benchmark), and a maximum speedup of $18\times$ for the context-sensitive configuration (on the **takl** benchmark) for the same number of workers. For DSS-FS, we achieve a maximum

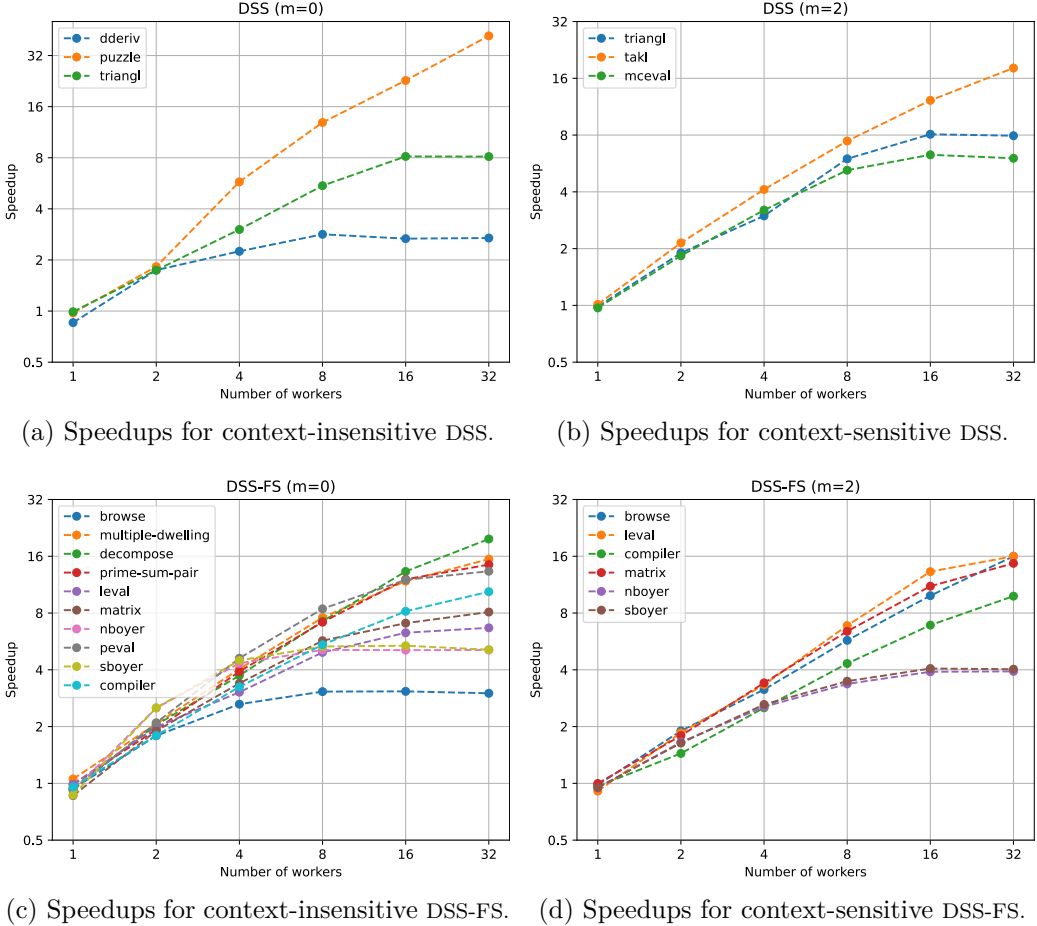


Figure 5.1: Speedups for DSS and DSS-FS, each in a context-insensitive and a context-sensitive ($m=2$) configuration for a varying number of workers.

speedup of $19\times$ for the context-insensitive configuration with 32 workers (on the `decompose` benchmark), and a maximum speedup of $16\times$ for the context-sensitive configuration (on the `leval` benchmark) for the same number of workers.

For the benchmarks we considered, speedups appear to be more dependent on the benchmark program rather than the configuration of the analysis (i.e., DSS vs. DSS-FS or context-insensitive vs. context-sensitive). Unsurprisingly, smaller programs with low running times in the sequential implementation show less potential for speedup. For context-insensitive DSS-FS, the `browse` benchmark is the shortest running program (taking only 1 second using the sequential implementation), and achieves noticeably worse speedups compared to the other benchmarks. Similarly, the `dderiv` benchmark for DSS takes less than 4 seconds using the se-

quential implementation, versus over 5 minutes for the **puzzle** benchmark. For context-sensitive DSS-FS, the same **browse** benchmark takes over 2 minutes using the sequential implementation, and shows consistent speedups up to 32 workers. One reason for this discrepancy is that our approach exploits data-parallelism in the worklist: larger worklist sizes offer more opportunities for parallelisation. A smaller, shorter running program may not be able to adequately saturate the worklist in order to take advantage of all available workers.

As expected, speedups are most consistent for a small number of workers: when using only 2 workers, our parallel worklist algorithm consistently achieves close to a $2\times$ speedup on most benchmarks. We notice that the speedup curve begins to “flatten” after 8-16 workers. One reason for this, as previously mentioned, is that benchmarks may struggle to consistently fill up the worklist with as many components as there are available workers (so that no workers are idling). Another reason is that for very large numbers of workers, our approach can become bottlenecked by the sequential coordinator: while multiple workers can analyse multiple components in parallel, the shared analysis state can only be updated by a single coordinator, the processing rate of which puts a limit on the maximum performance we can achieve (regardless of the number of workers).

These benchmarks also indicate that the overhead of our parallel worklist algorithm is low: when using only a single worker, the performance is very close to that of the sequential implementation. It should be noted, however, that the parallel worklist algorithm with 1 worker actually still uses 2 threads: one worker thread (which only analyses components) and one thread for the coordinator (which commits the results of each component analysis). This setup however rarely leads to actual speedups due to the poor work balance between both threads (i.e., the coordinator idles most of the time, as it only receives results from a single worker).

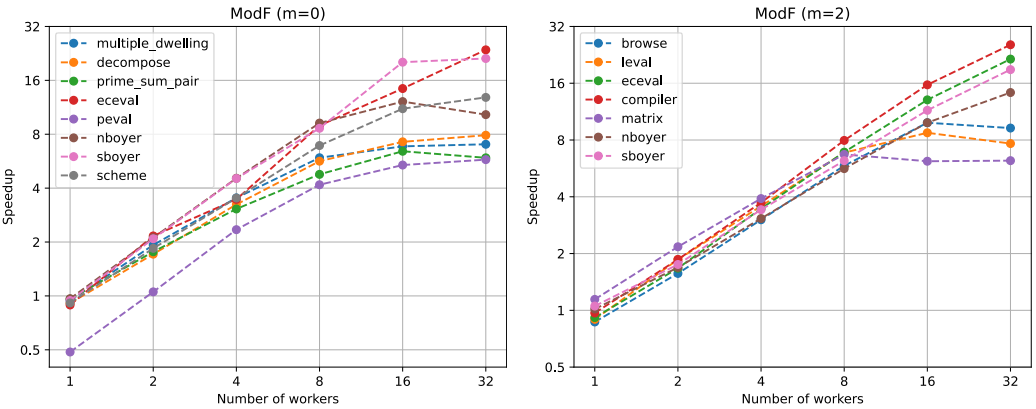
In rare cases, we may also observe superlinear speedups (e.g., for the **puzzle** benchmark in DSS). These can be explained by the fact that the total number of iterations that need to be performed by the analysis is dependent on the worklist exploration order. Parallelism adds non-determinism to this exploration order, as the relative rate at which workers analyse components determines which other components get scheduled, analysed and processed first. In addition, our parallel algorithm exploits the monotonicity of the analysis to speed up the convergence of the analysis state: every component analysis can update the analysis state, even if that component analysis was not using the latest analysis state itself. The increased rate at which the analysis state converges could in turn lead to fewer re-analyses of the same component, reducing the total amount of work for the parallel analysis. Indeed, for the **puzzle** benchmark, the sequential implementation requires a total of 140 251 iterations, whereas the parallel implementation

with 32 workers requires only a total of 79 464 iterations. This difference in the total amount of work that needs to be performed by the analysis, together with the use of 32 worker threads, explains the speedup of $41\times$. Our work confirms findings reported in related work [42, 4, 46, 93] that often attributes superlinear or unexpected speedups to the exploration order.

5.4.4 RQ3: Parallel Efficiency for ModF

To demonstrate that the general parallel worklist algorithm of Section 5.3 remains effective for analyses other than DSS, we also apply it to MODF, the modular analysis for which the effect-driven worklist algorithm of Nicolay et al. [119] was originally proposed. MODF is a function-modular analysis. Similar to DSS, its components correspond to (approximations of) function calls, so that an analysis of a single component corresponds to an intra-procedural analysis. Applying the parallel worklist algorithm (Algorithm 5) to MODF therefore results in an analysis where multiple function calls are analysed in parallel. Unlike DSS however, MODF is a flow-insensitive analysis, as it makes use of a single global store shared by all components. Therefore, unlike DSS, MODF also does not support abstract GC or abstract counting (and can therefore only perform weak updates).

Figure 5.2 shows the speedups for both context-insensitive (i.e., using $m = 0$) and context-sensitive (i.e., using $m = 2$) MODF when using our parallel worklist algorithm for a varying number of workers.



(a) Speedups for context-insensitive MODF. (b) Speedups for context-sensitive MODF.

Figure 5.2: Speedups for MODF, using both a context-insensitive and a context-sensitive ($m=2$) configuration for a varying number of workers.

Our observations here are similar to the observation we made in Section 5.4.3. We notice consistent speedups for MODF up to 16 workers. When using 8 work-

ers, most benchmarks achieve speedups between $4\times$ and $8\times$. When using 16 workers, most benchmarks achieve speedups between $4\times$ and $16\times$. Once more, longer running benchmarks (such as `eceval` and `compiler`) achieve the best speedups, whereas shorter running benchmarks (such as `prime_sum_pair` and `matrix`) struggle to maintain increasing speedups for a larger number of workers. Note that the pathological behaviour of the `peval` benchmark for context-insensitive MODF can again be attributed to the nondeterminism in the exploration order: the sequential implementation only needs to perform a total of 1899 analyses, whereas the parallel implementation with a single worker executes a total of 3100 analyses.

These results show that our parallel worklist algorithm is not DSS-specific: it is not just applicable to other analyses that employ the general worklist algorithm, but also has the potential to yield substantial speedups for these analyses.

5.5 Related Work

In this section, we discuss related work on sequential and parallel fixpoint algorithms. For both, we distinguish between algorithms that require some form of dependency graph a priori (such as a control flow graph (CFG)), and those where such dependencies can be discovered dynamically (i.e., during the analysis itself).

Sequential Fixpoint Algorithms (With Static Dependencies). Kildall [85] introduced one of the first worklist algorithms for computing dataflow analyses (presented in [85] for the purpose of program optimisation). Kildall’s algorithm keeps track of a set L of program points to process, and in each iteration removes and processes a single element from L (potentially adding new elements to L), repeating this loop until L is empty. In contrast to the worklist algorithm of Section 5.2, Kildall’s algorithm does not track dependencies dynamically to determine which program points need to be added back to L after each iteration; instead, it requires a *program graph* upfront, and upon processing a program point, adds all successor nodes of that program point in the graph to L if the analysis state changed (or equivalently — when expressed as a system of equations — all unknowns for which the data flow equation depends on the changed variable). Kildall points out that the order in which program points are selected from the set L impacts the number of iterations taken by the algorithm, but that no optimal selection method had been established — we observe the same sensitivity to the exploration order in our own experiments (Section 5.4.3).

When the CFG is known upfront, another simple, yet reasonably efficient fixpoint algorithm is to process all nodes in the CFG once in a fixed order, repeating this until the analysis state converges (also known as the *round-robin algorithm*).

Hecht and Ullman [64] show that when visiting nodes in reverse postorder, at most $d + 2$ iterations are required to reach convergence for bit-vector problems, where d is the largest number of back edges in any acyclic path in a reducible CFG. Kam and Ullman [81] later formulate the general *monotone dataflow framework* as a formal foundation through which many classical dataflow analyses can be expressed. They formulate the criteria an iteration sequence should adhere to in order to compute the least fixpoint¹⁰ of the monotone equations (both Kildall’s algorithm and the round-robin algorithm generate iteration sequences that meet these criteria), thereby also showing that the iteration order does not impact the analysis result when these criteria are met. Cooper et al. [31] later compare the round-robin algorithm to Kildall’s worklist algorithm in terms of performance (measured by the total number of iterations). They find that in general, with the right underlying worklist representation (e.g., a stack, queue or priority queue), the worklist algorithm’s exploration order outperforms the round-robin algorithm.

A well-known chaotic iteration strategy to compute fixpoints for dataflow analyses with known dependency graphs is Bourdoncle’s algorithm [23]. Bourdoncle’s algorithm decomposes the given dependency graph into a hierarchy of (strongly connected) components, and assigns each component a *head* element. It then computes a *weak topological ordering* (WTO) of the dependency graph, which is derived from this hierarchy. Unlike a regular topological ordering, a WTO relaxes the ordering condition so that it also imposes a total ordering of nodes for cyclic dependency graphs.¹¹ The iterative variant of Bourdoncle’s algorithm repeatedly processes all nodes once in the WTO order (similar to the round-robin algorithm). The recursive variant of Bourdoncle’s algorithm also follows the WTO order, but when encountering an SCC, first recursively computes the fixpoint for that SCC before continuing iteration. That is, the recursive variant respects the hierarchy of SCCs by first stabilising an inner SCC before continuing with the outer SCC. Bourdoncle also shows that the head of each SCC can be used as a widening point, and that the recursive and iterative variants may compute different solutions due to the non-monotonic nature of widening operators. In our setting, the domains we consider adhere to the ACC, and therefore do not require further widening.

Reps et al. [131] present IFDS, which offers an alternative approach to compute a solution for a class of analyses based on graph reachability. Specifically, the IFDS framework targets interprocedural analyses over a finite set D of dataflow facts with distributive transfer functions¹². IFDS reduces the analysis to a graph

¹⁰Kam and Ullman [81]’s presentation flips $\perp$ with $\top$ and the join operator ($\sqcup$) with the meet operator ($\sqcap$), thus technically computing the maximum (instead of least) fixpoint solution.

¹¹Specifically, for each edge $x \rightarrow y$ in the dependency graph, it requires that $x < y$ or that y is a head of an SCC containing x .

¹²Function f is distributive when $f(a \sqcup b) = f(a) \sqcup f(b)$ (whereas in general, $f(a \sqcup b) \supseteq f(a) \sqcup f(b)$).

reachability problem by constructing an exploded supergraph¹³: each node in the supergraph is expanded into $|D|+1$ nodes (one per dataflow fact plus one extra¹⁴), and an edge $(p_a, d_x) \rightarrow (p_b, d_y)$ is added whenever the transfer function of the edge $p_a \rightarrow p_b$ derives dataflow fact d_y from d_x . A dataflow fact d then holds at program point p when (p, d) is reachable from the initial node along a *realisable* path (i.e., one that at function boundaries always returns to the call site it entered from). A key aspect of IFDS that greatly improves its efficiency is its use of compact and reusable function summaries, which is enabled by the distributivity of the analysis. Distributivity holds for many analyses (such as live variable analyses), but not for the more general analyses considered in our own work.

Sequential Fixpoint Algorithms (Without Static Dependencies). Fixpoint algorithms for a system of equations where dependencies between unknowns are not known upfront naturally lend themselves to *top-down* solvers, as no fixed iteration order can be determined upfront. One of the most well-known solvers in this category is the **TD** algorithm given by Le Charlier and Van Hentenryck [92], originally proposed for the analysis of logic programs. The **TD** algorithm is a recursive-descent solver: when solving the equation f_X for X , it recursively solves the equation f_Y for each other unknown Y it encounters (keeping track of all unknowns on the call stack to avoid infinite recursion¹⁵). In contrast, a worklist algorithm, like the one presented in this chapter, uses the currently known value for each unknown Y it reads, and adds X back to the worklist for re-evaluation if the value of Y ever changes. Algorithm **TD** builds up a dependency map dynamically (i.e., during the fixpoint computation itself), mapping each unknown X to the set of other unknowns that are influenced by X (in order to *destabilise* all dependents of X when X changes). This is similar to how our own dependency map D maintains a mapping from dependencies to their dependent components.

Fecht and Seidl [47] present algorithm **W**, a generalisation of Kildall’s worklist algorithm where dependencies are discovered during the fixpoint computation itself. They present the worklist algorithm as a generic solver for the unknowns of a system of equations. Similar to algorithm **TD** and our own worklist algorithm, algorithm **W** maintains a dependency map from each unknown X to the set of other unknowns that are influenced by X , so that they can be added back to the worklist whenever X changes. Operationally, algorithm **W** is in essence

¹³The supergraph combines all functions’ CFGs, adding call/return nodes and linking edges.

¹⁴The extra node per program point serves as an artificial dataflow fact that always holds, so that it can be used as a source for unconditional dataflow facts.

¹⁵More precisely, the **TD** algorithm keeps track of a set of currently *Called* and *Stable* unknowns: it never recursively evaluates an unknown that is already called, and repeatedly evaluates unknowns until they have stabilised (i.e., only depend on other stable values).

identical to our own sequential worklist algorithm (and that of Nicolay et al. [119]). Fecht and Seidl also propose two improvements to algorithm **W**, resulting in algorithm **WRT** (and its sibling algorithm **WDFS**). The first improvement takes inspiration from algorithm **TD** by allowing a limited form of recursive descent: newly encountered unknowns are solved recursively instead of using an initial value $\perp$. This usually avoids an extra re-evaluation of the equation reading the newly encountered unknown once it gets assigned its initial non-bottom value. The second improvement assigns each worklist item a *timestamp*, and changes the worklist into a priority queue that processes items with a higher timestamp first. This ensures a more efficient scheduling in case of cyclic dependencies. Both improvements are complementary to our own work, and could therefore be integrated into the worklist algorithm presented in this chapter.

In this work, we build upon the worklist algorithm of Nicolay et al. [119], originally presented for MODF, a function-modular (flow-insensitive) analysis (having adapted it from a similar worklist algorithm for MODCONC, a thread-modular analysis [153]). MODF’s worklist algorithm is similar to that of algorithm **W**, but is formulated in terms of *components* and *effects* on a shared global store instead of directly being presented as a solver for a generic system of equations. That is, MODF uses *components* (approximations of function calls) as worklist items. The *inter-component* (i.e., inter-procedural) worklist algorithm processes each component using a simple *intra-component* (i.e., intra-procedural) analysis. Components can communicate (e.g., for argument and return value passing) through the shared global store, and the intra-component analysis logs all read and write *effects* to individual addresses of this store. Read effects are used to register dependencies of a component on addresses in the global store on the fly; write effects then trigger dependent components for re-analysis by adding them back to the worklist. Due to this global store, MODF analyses are inherently flow-insensitive. In this chapter, we have generalised this worklist algorithm by allowing the global store to be replaced with any other form of shared analysis state on which components can depend. This allows us to reuse the core of the algorithm for a wider class of analyses, such as the flow-sensitive DSS and DSS-FS. A key benefit of MODF’s (and MODCONC’s) worklist algorithm is the coarse granularity of components, which represent entire function executions (and thread executions). Our approach exploits this granularity for efficient parallelisation of the worklist algorithm.

In the original work on abstract definitional interpreters, Darais et al. [40] propose computing the fixpoint using a co-inductive caching algorithm. Their fixpoint algorithm is not worklist-based, but can be seen as a recursive descent solver (cf. sup.), as it recursively evaluates subexpressions directly. In order to break the recursion, the algorithm makes use of two caches in each iteration: an

input cache and an output cache. Each recursive evaluation of a call is cut by directly retrieving its result from the input cache, while the new result of that call is written to the output cache. The next iteration of the algorithm uses the output cache of the previous iteration as the new input cache, and the algorithm terminates once the output cache computed by an iteration is unchanged relative to the input cache it used. A downside of this approach is that each iteration needs to evaluate the entire program again (i.e., by evaluating the topmost expression), whereas our effect-driven fixpoint algorithm only re-evaluates configurations that were impacted by a change in the cache.

Keidel et al. [84] propose a modular approach to build fixpoint algorithms for big-step abstract interpreters (such as those of ADI and DSS). They argue that fixpoint algorithms for big-step abstract interpreters are hard to implement and reason about (e.g., for soundness proofs) as big monolithic entities, and instead introduce fixpoint *combinators* that can be composed to build fixpoint algorithms. This modularisation makes it easier to modify a single aspect of the fixpoint algorithm (by swapping one fixpoint combinator for another). The authors provide a library of fixpoint combinators, including φ_{topmost} , which mimics the fixpoint strategy of Darais et al. [40], as well as $\varphi_{\text{innermost}}$ and $\varphi_{\text{outermost}}$, which correspond to the recursive and iterative exploration strategies of Bourdoncle’s algorithm, respectively. In contrast to Bourdoncle’s algorithm, however, the resulting fixpoint algorithm does not precompute the iteration order from a given CFG, and instead discovers it during the fixpoint iteration itself. Keidel et al. also list the conditions a fixpoint algorithm for big-step abstract interpreters should adhere to in order to guarantee termination. These conditions are slightly more permissive than the ones we formulated for Algorithm 3: while we require the set of all possible configurations to be finite, they only require that every infinite *call chain* is eventually rendered finite (which they achieve through stack widening).

Parallel Fixpoint Algorithms (With Static Dependencies). Lee and Ryder [93] identified three types of parallelism in dataflow analysis: independent-problem parallelism, separate-unit parallelism, and algorithmic parallelism. Independent-problem parallelism arises when, conceptually, multiple analyses must be computed, and do not depend on each other. Separate-unit parallelism arises when the analysis can be performed on different components of the program: this is the kind of parallelism that we exploit in this work. Finally, algorithmic parallelism arises when the analysis of a single component is made parallel. The authors provide a parallel algorithm that divides each node of a (static) flow graph into

local computations, global propagation, and local propagation. The local phases can then be executed in parallel for each strongly-connected component of the flow graph. They achieve a speedup of up to $7.5\times$ on 8 cores for Fortran programs.

Kramer et al. [90] identify that previous approaches to parallelisation are limited by the structure of the program: the number of components that can be analysed in parallel greatly varies from program to program. They present an approach to solve classic dataflow problems (reaching definitions, live variables, available expressions, and very busy expressions) in parallel. Parallelism is achieved by identifying independent paths in the flow graph, partitioning linear paths, and by unrolling cyclic structures from the flow graph. In contrast to our own work, we focus on a more general class of analyses, and do not require a flow graph upfront.

Early work by Cousot [32] introduces the notion of *asynchronous iterations* as a theoretical framework for the parallel fixpoint computation of a system of equations. Asynchronous iterations generalise standard chaotic iterations: while a chaotic iteration only computes the equation for a single variable in each iteration step, asynchronous iterations allow for *multiple* variables to be solved in each iteration step (the set of unknowns that should be solved in each iteration is left open as a parameter of this framework). An important observation made by Cousot is that the asynchronous iteration at step k does not necessarily require the latest result for each variable (i.e., of step $k - 1$), but can in fact use the result of any previous iteration step $i < k$. This is exactly the same insight we exploit in our own parallel worklist algorithm, which still updates the analysis state after a component analysis if stale values of a dependency have been read.

Monniaux [114] describes the parallel implementation of the Astrée static analyser [37]. The parallelisation approach is tied to the analysis of low-level embedded C programs, where the analysed code contains many `switch` statements with many cases. Each case is therefore analysed on a separate thread, and a speedup of around $2\times$ is achieved on up to 5 cores. Cousot et al. [38] observed that Astrée does not benefit from parallelisation on more than 4 cores, as “the cost of synchronisation then outweighs the speedup of parallel execution”.

Xie and Aiken [181] present SATURN, a program analysis framework that is highly parallelisable, targeting C code in the Linux kernel. The high parallelisation opportunities come from the fact that SATURN performs bottom-up modular analysis: each function is analysed in isolation, and functions are analysed before their callers. This enables many independent functions to be analysed in parallel, resulting in speedups of up to $29\times$ on 80 cores.

Kim et al. [86] present a parallelisation of Bourdoncle’s fixpoint algorithm [23]. They generalise the concept of a weak topological ordering (WTO) into that of a *weak partial ordering* (WPO), so that elements that are independent in the partial

order can be computed in parallel. Their algorithm remains deterministic and computes the same result as Bourdoncle’s sequential algorithm (non-determinism may happen in a parallel abstract interpreter that relies on widening, which is not a monotone operation). This approach is well-suited for C programs, and the authors achieve up to an $11\times$ speedup on 16 threads.

Albarghouthi et al. [4] present an approach to parallelise top-down interprocedural modular analyses, which are less naturally parallelisable. Parallelisation is achieved through MapReduce-style parallelism: the map stage is used to analyse multiple functions in parallel, while the reduce stage manages inter-procedural dependencies. A speedup of up to $7.4\times$ is achieved on 8 cores.

For Java, Rodriguez and Lhoták [137] presented a parallelisation of the IFDS algorithm based on actors. Each CFG node is represented by an actor, and IFDS facts are propagated by sending messages between actors. A speedup of $6.12\times$ is achieved on 8 cores ($3.35\times$ compared to the baseline sequential algorithm).

Edvinsson et al. [46] parallelise a points-to analysis for Java programs by identifying *independent nodes* in the CFG, such as different targets of polymorphic calls, independent control-flow branches, and nodes with different contexts. A *task* performs the analysis of such an independent node, and is scheduled on a pool of worker threads. Synchronisation is required to continue the analysis after branching points. The overall goal is to find the right clustering of analysis nodes on the pool of worker threads for achieving an optimal analysis time. As this problem is NP-hard, heuristics are used by carefully selecting thresholds to ensure that each thread has sufficient work. They achieve a maximum speedup of $4.4\times$ on 8 cores.

Several approaches have been suggested to speed up an analysis by exploiting parallelism on the GPU. Mendez-Lojo et al. [102] implement parallel inclusion-based points-to analyses on GPUs. The analysis is made data-parallel by expressing it as graph rewrite rules: rules that do not interfere may safely be computed in parallel. They achieve a speedup of up to $3\times$ on 8 cores on a CPU, and of $7\times$ on a GPU. Prabhu et al. [129] present an algorithm for higher-order, context-insensitive (i.e., 0CFA) analysis for GPUs. The central idea is to encode the analysis data (syntax and abstract domains) as vectors and matrices, which benefit from high-performance operations on GPUs. The monotonicity of the analysis enables a less careful implementation, where data races or using stale analysis results do not compromise its soundness. Moving from a dense-matrix to a sparse-matrix representation of the problem (as CFGs tend to be sparse), the resulting analysis can achieve a speedup of up to $72\times$.

While our parallelisation approach exploits only thread-level parallelism on a single machine, other approaches achieve parallelism in a distributed setting. Venet and Brat [172] presented the first distributed static analysis implementa-

tion, focused on detecting out-of-bounds errors in embedded C programs. It relies on a relational database to store the analysis data, achieving speedups up to $2.7\times$ on 8 distributed CPUs. Two key design decisions are discussed that are also key in our own parallelisation approach: the granularity of the parallelism (which is set to analyse each function in a separate task) and the scheduling of the analysis tasks (for which a heuristic picks the most-called function first). They identified communication costs as the limiting factor for a distributed analysis, showing that these costs become too high beyond 4 distributed CPUs.

Parallel Fixpoint Algorithms (Without Static Dependencies). There has been little existing work on parallelising analyses for dynamic languages, where dependencies may not be known statically. An early effort by Weeks et al. [175] parallelised an abstract interpreter for a parallel higher-order dynamic language, by partitioning the state space of the analysis for different workers. Similar to our approach, each worker uses its own local state, which can be updated without synchronisation costs. They achieve a speedup of $9.4\times$ on 16 threads [42].

Dewey et al. [42] parallelise an abstract interpreter for JavaScript, by partitioning states per context. This is one of the few works that focuses on higher-order dynamic languages, where dependencies may not be known statically. The speedups they report range from $2\times$ to $4\times$ on 12 threads, with a few outliers up to $37\times$. In contrast, our approach does not need to partition components per context, and is therefore also applicable to context-insensitive analyses.

Recent work by Kocal et al. [89] also proposes a novel parallel fixpoint algorithm for equation systems where dependencies between variables are not known upfront. The main difference with our approach is that their algorithm is not based on the worklist solver **W**, but instead extends the recursive descent solver **TD** (cf. sup.), resulting in a novel parallel solver **TLTD**. Their approach focuses on enabling the parameterisation of the level of granularity for parallel tasks. That is, an instantiation can choose a set of root variables that are each mapped to a worker thread: a single worker for a given root variable then also solves all unknowns that are recursively descended into when solving the equation for that root variable. As such, one can mimic our parallelisation approach for DSS(-FS) and MODF, which all use a separate worker for each function call, by selecting the variables corresponding to function entry points as root variables. Interestingly, this work also explores two implementation strategies to allow for communication between workers: one based on a thread-safe shared state that is immediately accessed by all workers, and one where all workers maintain analysis state locally and communicate through a publish-subscribe protocol. The latter roughly corresponds to the strategy used in our own parallel implementation of the worklist

algorithm. The speedups reported by the authors are promising: with 8 workers, most benchmarks achieve speedups in the $2\text{--}8\times$ range, similar to what we observe for our own algorithm. It would therefore be interesting to explore the use of the parallel **TLTD** solver for DSS-based analyses in future work, so that it can be directly compared to our own worklist-based parallelisation approach.

5.6 Conclusion

In this chapter, we have generalised an existing effect-driven worklist algorithm, so that it can be instantiated as a fixpoint algorithm for DSS and DSS-FS. By subsequently parallelising this algorithm, we present a novel approach to construct parallel analyses for dynamic, higher-order languages (including but not limited to parallel versions of DSS and DSS-FS). The key insight in our work is that this worklist algorithm offers inherent opportunities for efficient parallelisation, which we exploit by analysing different components in parallel. Despite its non-deterministic behaviour, this algorithm obtains the exact same result as the sequential worklist algorithm, and is able to further exploit the monotonicity of the analysis to speed up its convergence.

We applied this parallel worklist algorithm to both DSS and DSS-FS, as well as MODF, an existing function-modular analysis. Our implementation in the MAF framework reveals significant speedups. For all considered instantiations, we observe consistent speedups up to 16 workers (usually between $4\times$ and $16\times$). As such, in general our parallelisation strategy appears to achieve similar or better speedups compared to existing parallel analyses. In addition, it does not require the control-flow graph of the program under analysis beforehand, and can therefore also be used for analyses that target dynamic, higher-order languages. Finally, it is directly applicable to both context-sensitive and context-insensitive analyses, and to analyses both with and without abstract GC. The latter allows us to stack the performance benefits of abstract GC with those of our parallelisation algorithm, therefore attacking scalability issues on two fronts at the same time.

6 Conclusion and Future Work

Static analysis enables reasoning about the run-time behaviour of a program without executing that program. It is the foundation for program verification, comprehension and optimisation. In order to be considered useful in practice, static analyses require both adequate precision and performance. Unfortunately, achieving both at the same time can prove challenging, as most techniques that increase precision come at the cost of performance (and vice versa).

In this work, we have presented various techniques that can improve both the precision and performance of static analyses at the same time. These efforts were twofold. First, we focused on abstract garbage collection, which is known to benefit both precision and performance on its own. However, many of these benefits are often lost in its traditional integration into modern abstract interpreters, partly due to its performance overhead, and partly due to its incompatibility with big-step abstract interpreters. We introduced novel techniques, *abstract reference counting* (Chapter 3) and *delta store semantics* (Chapter 4), that overcome these issues, so that both the precision and performance benefits of abstract GC are maximised. Second, we further improved performance by proposing an efficient, parallel fixpoint algorithm for such analyses (Chapter 5).

In this concluding chapter, we provide a brief summary of this dissertation, recapitulate our contributions and provide guidance for future work.

6.1 Summary of the Dissertation

In Chapter 1, we motivated the importance of abstract GC, highlighting the mechanisms through which it improves the precision and performance of an abstract interpreter: avoiding garbage-induced imprecision (e.g., by preventing joins with garbage values) and rendering analysis results more reusable (through garbage irrelevance). We then focused on two problems faced by existing approaches to abstract GC: its performance overhead, caused by the need for frequent applications that traverse the entire heap (which partly negates abstract GC’s performance benefits), and its poor compatibility with big-step abstract interpreters (due to the lack of an explicit continuation from which to extract root addresses).

In Chapter 2, we introduced a minimal higher-order language λ_{ANF} , which is used as a basis for all formalisations in this dissertation. We presented the small-step concrete and abstract semantics of λ_{ANF} (following the Abstracting Abstract Machines (AAM) approach [105]), and showed how (abstract) GC is traditionally integrated into these (abstract) semantics by using a tracing collector to reclaim unreachable addresses at every evaluation step. We also highlighted the well-known synergy between abstract GC and abstract counting: abstract GC can keep abstract counts lower (by resetting counts for reclaimed addresses to 0), while abstract counting can allow for more garbage to be reclaimed (by removing values from the store through strong updates).

In Chapter 3, we tackled a first issue of the traditional integration of abstract GC presented in Chapter 2: GC is applied at every evaluation step, which leads to a significant performance overhead (therefore partly negating the performance benefits of abstract GC) as the entire heap needs to be traversed at every GC application. At the same time, applying GC less frequently would no longer eliminate all garbage-induced imprecision (therefore partly negating the precision benefits of abstract GC). We surveyed existing literature and implementations, and found that abstract GC is often applied only at specific program points to avoid this issue, consciously sacrificing precision for performance. In order to avoid having to make this tradeoff, we subsequently proposed a novel alternative approach to collect all garbage in an abstract interpreter based on reference counting instead of tracing GC, which we referred to as *abstract reference counting*. By using abstract reference counting, garbage is immediately reclaimed using simple book-keeping. Frequent heap traversals are avoided altogether, which makes abstract reference counting insensitive to the size of the heap: the cost of collecting garbage is only proportional to the amount of garbage being collected. We formalised the small-step abstract semantics augmented with abstract reference counting. We then pointed out that naive abstract reference counting does not suffice to keep the interpreter garbage-free: reference counting cannot reclaim garbage cycles, and we showed that cyclic structures are more common in an abstract interpreter due to abstraction. Therefore, we presented an extended version of abstract reference counting with cycle detection: by detecting all cyclic structures (SCCs) and counting references at the level of SCCs, we overcame this limitation of reference counting and were able to reclaim all garbage. While our cycle detection algorithm follows a naive approach to detect cycles, we provided both formal and empirical evidence that it is reasonably efficient for an abstract interpreter. Our experimental evaluation showed that abstract reference counting with cycle detection indeed achieves exactly the optimal precision (i.e., the same as applying GC at every step), while exhibiting an order-of-magnitude performance improvement over existing approaches that rely on tracing GC.

In Chapter 4, we tackled a second issue of traditional integrations of abstract GC: big-step abstract interpreters (e.g., those following the Abstracting Definitional Interpreters (ADI) approach [40]) offer several advantages compared to the small-step abstract interpreters presented in Chapter 2 (and Chapter 3), but are not immediately compatible with abstract GC. The problem is that abstract GC requires a root set of addresses, which traditionally includes addresses reachable from the continuation; this continuation is no longer directly accessible in a big-step abstract interpreter. We argued that existing approaches to combining abstract GC with big-step abstract interpreters sacrifice some of the benefits of either technique and/or greatly overcomplicate the abstract semantics. We therefore introduced an alternative form of big-step semantics that allows for a more efficient integration of abstract GC, which we referred to as *delta store semantics*. Delta store semantics remain close to the original big-step semantics, but return a *delta store* representing all changes to a store instead of the full updated store. We first formalised DSS without abstract GC, and then showed how abstract GC can easily and efficiently be integrated into these semantics by adding a single evaluation rule. DSS with abstract GC preserves the full precision and performance benefits of abstract GC and big-step abstract interpreters. In addition, it unlocks an extra synergistic benefit in the form of *stackless GC*: DSS can collect garbage without continuation addresses as part of the root set, leading to more garbage being collected and therefore amplifying the positive effects of abstract GC. We presented a formal claim, along with a mechanised proof in Rocq (for programs without mutable variables), showing that this leads to abstract GC achieving superior precision in DSS compared to its small-step (AAM) counterpart. The results of our experimental evaluation are consistent with this claim. However, they also show that DSS still suffers from scalability issues, due to the combinatorial explosion of program states it has to explore in case of finite abstract domains, and potentially infinite number of states for infinite abstract domains. In order to overcome these scalability and termination issues, we also presented DSS-FS, a version of DSS with flow-sensitive store widening. Store widening avoids the state space blow-up, thereby significantly improving performance in general. Unlike traditional (global) store widening, DSS-FS’s flow-sensitive store widening only widens per program point, and therefore remains compatible with abstract GC and abstract counting. Our experiments showed that DSS-FS overcomes the scalability issues of DSS (resolving 15 out of the 18 timeouts in our benchmarks), at the cost of giving up some of the precision benefits of abstract GC.

In Chapter 5, we presented a fixpoint algorithm to efficiently compute a sound solution using the evaluation rules given in Chapter 4. We generalised an existing effect-driven worklist algorithm that has already proven effective for a class

of flow-insensitive, modular analyses. We then presented instantiations of this generalised worklist algorithm for DSS and DSS-FS. These instantiations break the recursion in the evaluation rules presented in Chapter 4 by passing around a cache, therefore showing how such analyses can be implemented in practice. Subsequently, we presented a parallel version of this generalised worklist algorithm, therefore automatically rendering all instantiations of this algorithm (such as those of DSS and DSS-FS) parallel “for free”. Our experimental evaluation showed that this parallel version scales well for our benchmark programs on multi-core CPUs for both DSS(-FS) and other analyses that can be expressed using the generalised worklist algorithm (achieving consistent speedups up to 16 workers, with a maximum observed speedup of $41\times$ when using 32 workers).

Overview of Presented Approaches and Their Characteristics. Throughout the dissertation, we have discussed several approaches to abstract interpretation, some from existing work and some newly developed in this work. Table 6.1 provides an overview of these approaches, summarising their characteristics with respect to abstract GC, pushdown CFA, flow sensitivity, scalability and requirements on the abstract domain. We do not include the worklist algorithms of Chapter 5 in this overview, as the choice of fixpoint algorithm is orthogonal to these approaches.

Abstract reference counting, which we introduced in Chapter 3 for a small-step (AAM) abstract interpreter, is the only approach to abstract GC with a low GC overhead. All approaches that apply abstract GC enjoy its precision (and the corresponding performance) benefits. For DSS-FS, however, this effect is muted, as some (but not all) of the precision benefits are negated due to the store widening. Due to its use of a global store, MODF cannot perform GC at all, and crucially also does not support abstract counting (and therefore strong updates). All approaches that apply abstract GC also enjoy the benefit of garbage irrelevance. MODF can also be considered to enjoy garbage irrelevance: even though it does not apply abstract GC, garbage in the global store of MODF does not lead to multiple redundant evaluations of the same component. DSS and DSS-FS are the only approaches that unlock the extra benefit of stackless GC, which forms a core advantage of these approaches compared to existing work.

AAM-based approaches feature none of the advantages of pushdown CFA.¹ In contrast, big-step abstract interpreters inherit perfect stack precision and context irrelevance for free. However, the ADI approach combined with abstract GC (as presented in [40]) gives up context irrelevance in order to support abstract GC, whereas DSS and DSS-FS do so without giving up this desirable property. ModF

¹Variants of AAM, such as AAC [57] and P4F [56], do enjoy perfect stack precision (by construction), but not context irrelevance. We omit these variants from Table 6.1 for brevity.

First Appearance	Abstract GC				Pushdown CFA				
	Low GC Overhead	GC Improves Precision	Garbage Irrelevance	Stackless GC	Perfect Stack Precision	Context Irrelevance	Flow Sensitivity Scalability	Terminates Under ACC	
<u>AAM</u>									
without abstract GC [105] (§2.3)	n/a	×	×	n/a	×	×	✓	×	✓
with abstract GC (FCFA) [105, 106] (§2.3.2)	×	✓	✓	×	×	×	✓	×	×
with ARC §3.2	✓	✓	✓	×	×	×	✓	×	×
<u>ADI</u>									
without abstract GC [40] (§4.2)	n/a	×	×	n/a	✓	✓	✓	×	✓ [†]
with abstract GC [40] (§4.2)	×	✓	✓	×	✓	×	✓	×	×
DSS §4.3	×	✓	✓	✓	✓	✓	✓	×	×
DSS-FS §4.5.2	×	~	✓	✓	✓	✓	✓	✓	✓
ModF [119]	n/a	×	✓	n/a	✓	✓	×	✓	✓

Table 6.1: Overview of the approaches discussed in this dissertation and their characteristics (✓ = yes, × = no, ~ = partially, n/a = not applicable). Approaches highlighted in **bold** are those developed in this work. [†]Assuming abstract counting is also disabled.

follows the design of big-step abstract interpreters (without abstract GC), and therefore also enjoys both perfect stack precision and context irrelevance. The advantages of pushdown CFA come at a cost not visible in Table 6.1: approaches that do not model the continuation (ADI, DSS, DSS-FS and ModF) do not support first-class continuations directly, whereas they can be supported trivially for small-step abstract interpreters (such as AAM).

ModF is the only analysis in Table 6.1 that does not support flow sensitivity, again due to its use of a global store. Therefore, it can be considered less precise than the other analyses (such as DSS-FS) even without the use of abstract GC.

We consider all approaches that employ local stores (i.e., stores that are directly part of states and configurations) as not scalable, since local stores directly lead to exponential analysis complexity. Therefore, only DSS-FS, which widens stores per program point, and ModF, which only maintains a single global store, are marked as scalable. Our experiments indeed confirm that only these approaches are able to consistently avoid timeouts for larger programs.

The last column indicates the requirement on the abstract domain $\widehat{\text{Val}}$ to guarantee termination for the abstract interpreter. Without abstract GC and strong updates (through abstract counting), stores can only grow through joins, and therefore form ascending chains on each execution path so that the ACC for $\widehat{\text{Val}}$ suffices to guarantee convergence (and therefore termination). In Table 6.1, this is the case for the AAM-based and ADI-based abstract interpreters without abstract GC. With abstract GC and/or strong updates (through abstract counting), which remove bindings from the store, this monotonicity guarantee breaks, and therefore a finite domain is required to guarantee termination. In Table 6.1, this is the case for ΓCFA , AAM with ARC, ADI with abstract GC, and DSS. Store widening can recover termination under the ACC, as an ascending chain of stores is obtained by construction. In Table 6.1, this is the case for DSS-FS and MODF. Note however that the global store widening of MODF is incompatible with abstract GC; DSS-FS is therefore the only technique that supports both abstract GC (and strong updates) while only requiring the ACC for its abstract domain.

6.2 Contributions

We restate the main contributions of the work presented in this dissertation.

Abstract Reference Counting. In Chapter 3, we presented a novel approach to avoid the performance overhead of abstract GC by using reference counting instead of tracing GC techniques. Our resulting approach is able to collect all garbage, therefore achieving the optimal precision with only minimal overhead. Concretely, we made the following contributions in this chapter:

- We introduced abstract reference counting as a more efficient alternative to abstract tracing GC, providing a formalisation of abstract reference counting for a small-step (AAM-based) abstract interpreter.
- We proved that abstract reference counting is not only GC sound (i.e., it only removes garbage), but also garbage-free (i.e., it removes all garbage) in the absence of cyclic pointer chains in the heap.
- We identified sources of garbage cycles during abstract interpretation, and presented a novel cycle detection algorithm for abstract reference counting based on these observations. We showed that per insertion to the abstract interpreter’s continuation store, where cyclic garbage causes the most issues, the algorithm only requires amortised $\mathcal{O}(1)$ additional operations.
- We provided an implementation of abstract reference counting in the Scala-AM framework² and an empirical evaluation of our approach using the Gab-

²Publicly available at <https://github.com/noahvanes/scala-am-abstractgc>.

riel benchmark suite for Scheme programs [49]. The results showed that our approach achieves optimal precision and is significantly more performant compared to existing approaches to abstract GC.

Delta Store Semantics. In Chapter 4, we introduced DSS, a novel formulation of abstract big-step semantics that allows for an efficient integration of abstract GC. In addition, we tackled scalability issues for abstract interpreters using local stores by introducing a form of store widening for DSS that remains compatible with abstract GC. Concretely, we made the following contributions in this chapter:

- We presented DSS as a novel formulation of big-step (ADI-style) abstract semantics where evaluation steps return *minimal* and *composable* delta stores (representing changes to a store) instead of complete updated stores.
- We showed how abstract GC can be integrated into DSS, exploiting the full potential of combining abstract GC with pushdown control flow. Thanks to stackless GC, this integration achieves better precision than traditional abstract GC for small-step (AAM-like) abstract interpreters. We formulated this claim as a theorem, and provided a mechanised proof for it in Rocq for programs that do not make use of variable assignments.
- We introduced flow-sensitive store widening for DSS, further exploiting the minimal and composable nature of delta stores. Unlike global store widening, this form of store widening remains compatible with abstract GC.
- We provided an implementation of DSS (including the integrations of abstract GC and flow-sensitive store widening) in the MAF framework³. Using this implementation, we conducted an evaluation that provided empirical evidence for the previously stated precision claim and for the benefit of flow-sensitive store widening for the scalability of the analysis.

A Parallel Worklist Algorithm. In Chapter 5, we introduced an efficient, parallel worklist algorithm to compute the fixpoint for a wide class of analyses (including but not limited to analyses based on DSS and DSS-FS). Concretely, we made the following contributions in this chapter:

- We generalised an existing effect-driven worklist algorithm, so that it can be used as a (sequential) fixpoint algorithm for a wider class of analyses.
- We instantiated this generalised worklist algorithm for DSS and DSS-FS, thereby showing how such analyses can be implemented efficiently.
- We proposed a novel parallel version of this generalised worklist algorithm, automatically rendering all of its existing instantiations parallel for free.

³Publicly available at <https://github.com/softwarelanguageslab/maf>.

- We integrated this parallel worklist algorithm into MAF⁴ and conducted an evaluation showing that it can achieve significant speedups on multi-core CPUs for both DSS(-FS) and other instantiations of the worklist algorithm.

6.3 Limitations & Future Work

In this section, we propose several directions for future work, based on limitations of the current work presented in this dissertation.

DSS with Abstract Reference Counting. We have introduced abstract reference counting and DSS independently of one another in Chapter 3 and Chapter 4, respectively. That is, our current formulation of DSS still uses abstract tracing GC to collect garbage, and therefore suffers from a significant performance overhead as GC is applied at every step⁵ (as indicated in Table 6.1). In order to overcome this issue, we could replace the tracing GC of DSS with abstract reference counting. We argue that integrating abstract reference counting into DSS is mostly a technical effort. A simple approach could, for instance, only count references for every store $\hat{\sigma}$; whenever a store is modified by applying a delta store, the reference map $\hat{\phi}$ can then be updated accordingly for $\hat{\sigma}$. As such, only the input configurations of DSS need to be extended to carry an additional component $\hat{\phi}$ to correctly maintain the bookkeeping for abstract reference counting.

Alternative Store Widening Strategies for DSS-FS. In Chapter 4, we introduced DSS-FS, a variant of DSS with flow-sensitive store widening to solve its scalability issues while preserving compatibility with abstract GC. While DSS-FS remains compatible with abstract GC, it arguably loses too many of the precision benefits that abstract GC offers due to widening. As discussed in Section 4.5.3, the root cause of the problem is that only a single store is maintained per program point, so that all allocations at that program point are joined in the same store. In Section 4.7, we have already hinted at several less aggressive store widening strategies that would preserve more of abstract GC’s precision benefit. Trace partitioning [99, 113], for instance, would keep multiple stores per program point based on different abstract traces. Another promising alternative to our own flow-sensitive store widening for DSS-FS is the stack widening principle used by Keidel et al. [84], which only joins stores for program points that occur in a loop. Under such a stack widening strategy, the precision benefits of abstract GC would only

⁴Publicly available at <https://github.com/softwarelanguageslab/maf/tree/parallel-dss>.

⁵In fact, to avoid the impact of this overhead on our measurements, we only applied abstract GC for DSS at every function call for the experiments conducted in Chapter 5.

be sacrificed locally when needed to avoid infinite call chains. Finally, adaptive widening techniques, such as those proposed by Oh et al. [124], could be used to apply store widening for only some addresses in the store, so that non-widened addresses still obtain the maximum benefit of abstract GC.

Support for First-Class Continuations. In this work, we used a minimal language λ_{ANF} as a target for our analyses. Neither λ_{ANF} nor the subset of R5RS supported by our implementations features first-class continuations (such as those captured by `call/cc`). The impact of this restriction differs per contribution:

- The restriction is not inherent to abstract reference counting. In Chapter 3, we presented abstract reference counting for an AAM-based abstract interpreter. For such small-step interpreters, adding support for first-class continuations is straightforward, as continuations are modelled explicitly, and continuation addresses can be used as a representation for continuation values. Abstract reference counting would remain garbage-free and GC sound, as Theorem 8 makes no assumptions about the references between the value and continuation stores. The amortised $\mathcal{O}(1)$ bound of Theorem 7 is no longer guaranteed, however, as discussed in Section 3.3.3.
- For DSS (Chapter 4), the restriction is inherent to its foundation in big-step abstract interpreters, which do not explicitly model the continuation. The challenge lies only in supporting re-entrant continuations (such as those captured by `call/cc`) and language features that suspend and resume computations (e.g., generators and coroutines in Python or JavaScript). In contrast, non-local control that escapes to an enclosing context (e.g., through continuations captured with `call/cc`, and through language features such as exceptions, `break` and `return`) can directly be supported in big-step abstract interpreters (including DSS) by modelling it as an additional evaluation result that is propagated to that enclosing context. For re-entrant continuations, using a CPS transformation for code that makes use of first-class continuations would in turn void the main benefits of pushdown control flow for that code (as discussed in Section 4.1.1). An alternative approach, as demonstrated in [180], is to exploit the cache of the abstract interpreter: in ADI-based analyses, one can return a value to the continuation of any procedure call by writing that value to the cache entry for that call (i.e., for the corresponding configuration). Doing so directly enables support for first-class continuations, and therefore also other non-local control constructs. For DSS, this would additionally require computing the delta store for that configuration, given the store that the continuation was called with. We leave such an extension of DSS open as future work.

- The parallel worklist algorithm of Chapter 5 is orthogonal to this restriction. It is parameterised by the choice of components, dependencies and component analysis, and only inherits the restriction through its instantiations (for DSS, DSS-FS and MODF). It already supports the mechanism described above to extend DSS with first-class continuations, as the analysis of a component may update arbitrary dependencies (in this case, the result for the cache entry of the configuration whose continuation is invoked).

Support for Richer Abstract Domains. In this work, we have made several assumptions on the abstract domain used by our analyses. For DSS, we support only finite domains, whereas for DSS-FS, we require that the abstract domain adheres to the ACC. In order to build more interesting analyses on top of our techniques, we argue that our analyses can be extended with widening operators [33] to support domains with infinite height lattices (such as the well-known interval domain). These widening operators could for instance be (selectively) inserted as a replacement for joins when merging stores per program point in DSS-FS. In order to avoid losing too much precision prematurely, ideally this should be combined with a stack widening policy, as in the work of Keidel et al. [84]. In addition, all domains considered in this work are non-relational; relational domains such as polyhedra [36] and octagons [110] (e.g., from the Apron library [72]) are not supported. Abstract GC can be supported for relational domains through *projection*: relational variables that are associated with a collected address are existentially quantified out of the relational state. An open question is how the representation of delta stores can be extended to represent changes to both the non-relational and relational part of the abstract state, and how operations on delta stores (such as composition and replay) need to be adapted accordingly.

Improved Incremental Cycle Detection for Abstract Reference Counting. Abstract reference counting requires an incremental cycle detection algorithm that can handle both the addition and the deletion (due to strong updates) of edges. For this purpose, we currently employ a simple algorithm, which we have shown to perform reasonably well for detecting cycles in an abstract heap. However, more advanced cycle detection algorithms have been proposed [136, 61, 14] that promise better asymptotic performance compared to our own approach. While our benchmarks currently do not yet show a significant performance overhead from cycle detection, other programs and/or abstractions that lead to less “shallow” pointer structures in the heap may benefit from adopting these algorithms.

Parallelisation of Other Fixpoint Algorithms. In Chapter 5, we have parallelised a worklist-based fixpoint algorithm. Worklist algorithms offer inherent opportunities for parallelisation, as items in the worklist can be processed concurrently. However, as discussed in Section 5.5, alternative fixpoint algorithms often prove more efficient in practice. One possible avenue of future work is to integrate the improvements proposed by Fecht and Seidl [47] for their sequential worklist algorithm into our structurally equivalent parallel worklist algorithm. Another possible direction is to investigate the parallelisation of solvers that are not worklist-based, but instead rely on recursive descent (such as those in [23, 92, 84]). Kim et al. [86] already present a parallel version of Bourdoncle’s algorithm, but their technique requires the iteration strategy to be precomputed from the CFG. Keidel et al. [84] present a sequential version that follows Bourdoncle’s iteration strategy without requiring a CFG upfront. It remains an open question whether we can combine the best of both techniques, so that we obtain a parallel, recursive-descent fixpoint solver that does not require a dependency graph upfront.

6.4 Concluding Remarks

Software has become too complex and too present in today’s society to manage without the help of automated tools that can reason about these software systems. Static analysis is the foundation of these tools. However, static analysis is only truly useful in practice when it meets certain standards in terms of precision and performance. Unfortunately, most available techniques to increase the precision of the analysis come at the cost of performance, and vice versa, so that an analysis that is both fast and precise is hard to come by. There are, however, a few techniques that manage to improve both precision and performance at the same time. In this work, we have focused on two such techniques — abstract GC and parallelisation — with the aim of getting the most out of them.

We argue that abstract GC is a key enabler for scalability, as larger programs bring about larger heaps that require localised reasoning with compact cached analysis results (in an ideal setting: purely local summaries, which are the foundation for scalability in, for instance, separation logic). Our work renders abstract GC more applicable to existing abstract interpreters without compromise. It shows that abstract GC can be made cheap to integrate (thanks to abstract reference counting), removing the need to choose between the precision and performance benefits of abstract GC. It shows that abstract GC can be made compatible with modern big-step abstract interpreters (thanks to delta store semantics), removing the need to sacrifice the benefits of either. At the same time, our empirical evaluations also show that the analysis designs where abstract GC brings about

its greatest benefits cannot themselves be considered scalable (mostly due to the use of local stores without any form of store widening). Our work therefore explored a store widening technique that does not fully compromise the benefits of abstract GC. We argue that finding a balance between such widening techniques (e.g., using the stack widening principle) and abstract GC is key to further maximising overall analysis scalability and precision.

Parallelisation is another orthogonal approach we proposed in our work to further speed up the analysis without compromising its precision. By parallelising a general worklist algorithm, we argue that our parallelisation approach can be used to parallelise a large class of analyses for dynamic, higher-order languages.

The different approaches presented in this work — abstract reference counting, delta store semantics, and the parallel worklist algorithm — can all be considered orthogonal to each other. Therefore, they can all be used independently of one another, or be combined as needed to get the benefits of each approach.

Bibliography

- [1] Adams, N. I., Bartley, D. H., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., Steele, G. L., Sussman, G. J., Wand, M., and Abelson, H. (1998). Revised⁵ report on the algorithmic language Scheme. *SIGPLAN Notices*, **33**(9), 26–76.
- [2] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1983). *Data Structures and Algorithms*. Addison-Wesley.
- [3] Al Madi, N. (2022). How readable is model-generated code? examining readability and visual inspection of GitHub Copilot. In *Proceedings of the 37th IEEE ACM International Conference on Automated Software Engineering*, pages 205:1–205:5, Rochester, Michigan, USA. ACM.
- [4] Albarghouthi, A., Kumar, R., Nori, A. V., and Rajamani, S. K. (2012). Parallelizing top-down interprocedural analyses. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 217–228, Beijing, China.
- [5] Alrashedy, K., Aljasser, A., Tambwekar, P., and Gombolay, M. (2025). Leveraging static analysis for feedback-driven security patching in LLM-generated code. *Journal of Cybersecurity and Privacy*, **5**(4), 110.
- [6] Amorim, E. V., Fernandes, A., and Filipe, V. (2024). Analyzing the impact of the CrowdStrike tech outage on airport operations and future resilience strategies. In M. M. Cruz-Cunha, D. Domingos, E. Peres, and R. Rijo, editors, *Proceedings of the International Conference on Enterprise Information Systems / International Conference on Project Management / International Conference on Health and Social Care Information Systems and Technologies*, volume 256 of *Procedia Computer Science*, pages 633–640, Funchal, Madeira Island, Portugal. Elsevier.
- [7] Andreasen, E. S., Møller, A., and Nielsen, B. B. (2017). Systematic approaches for increasing soundness and precision of static analyzers. In K. Ali and C. Cifuentes, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 31–36, Barcelona, Spain. ACM.

- [8] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Outeau, D., and McDaniel, P. D. (2014). FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In M. F. P. O’Boyle and K. Pingali, editors, *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, Edinburgh, United Kingdom. ACM.
- [9] Assaf, M., Naumann, D. A., Signoles, J., Totel, E., and Tronel, F. (2017). Hypercollecting semantics and its application to static analysis of information flow. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 874–887, Paris, France. ACM.
- [10] Bacon, D. F. and Rajan, V. T. (2001). Concurrent cycle collection in reference counted systems. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 207–235, Budapest, Hungary.
- [11] Bacon, D. F., Cheng, P., and Rajan, V. T. (2004). A unified theory of garbage collection. In J. M. Vliissides and D. C. Schmidt, editors, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 50–68, Vancouver, BC, Canada.
- [12] Baker Jr., H. G. (1978). List processing in real time on a serial computer. *Communications of the ACM*, **21**(4), 280–294.
- [13] Balakrishnan, G. and Reps, T. W. (2006). Recency-abstraction for heap-allocated storage. In K. Yi, editor, *Proceedings of the 13th International Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239, Seoul, Korea. Springer.
- [14] Bender, M. A., Fineman, J. T., Gilbert, S., and Tarjan, R. E. (2016). A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms*, **12**(2), 14:1–14:22.
- [15] Berdine, J., Calcagno, C., and O’Hearn, P. W. (2005). Symbolic execution with separation logic. In K. Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68, Tsukuba, Japan. Springer.
- [16] Bergstrom, L. (2009). *Arity raising and control-flow analysis in Manticore*. Master’s thesis, University of Chicago.

- [17] Bergstrom, L., Fluet, M., Le, M., Reppy, J. H., and Sandler, N. (2014). Practical and effective higher-order optimizations. In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 81–93, Gothenburg, Sweden. ACM.
- [18] Binkley, D. W., Raszewski, L. R., Smith, C., and Harman, M. (2000). An empirical study of amorphous slicing as a program comprehension support tool. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 161–170, Limerick, Ireland. IEEE Computer Society.
- [19] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In R. Cytron and R. Gupta, editors, *Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, San Diego, California, USA. ACM.
- [20] Blazy, S., Pichardie, D., and Trieu, A. (2019). Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, **27**(1), 137–163.
- [21] Boehm, H. (2004). The space cost of lazy reference counting. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 210–219, Venice, Italy.
- [22] Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Ghorbal, K., Goubault, E., Lesens, D., Mauborgne, L., Miné, A., *et al.* (2009). Space software validation using abstract interpretation. In *Proceedings of the International Space System Engineering Conference: Data Systems in Aerospace*, volume 1, pages 1–7. European Space Agency.
- [23] Bourdoncle, F. (1993). Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, pages 128–141, Akademgorodok, Novosibirsk, Russia.
- [24] Brat, G. and Venet, A. (2005). Precise and scalable static program analysis of NASA flight software. In *Proceedings of the IEEE Aerospace Conference*, pages 1 – 10.
- [25] Calcagno, C., Distefano, D., O’Hearn, P. W., and Yang, H. (2011). Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, **58**(6), 26:1–26:66.

- [26] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P. W., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. (2015). Moving fast with software verification. In K. Havelund, G. J. Holzmann, and R. Joshi, editors, *Proceedings of the 7th International NASA Formal Methods Symposium*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11, Pasadena, California, USA. Springer.
- [27] Chen, L. and Harrison III, W. L. (1994). An efficient approach to computing fixpoints for complex program analysis. In J. R. Gurd and W. Jalby, editors, *Proceedings of the 8th international conference on Supercomputing*, pages 98–106, Manchester, United Kingdom. ACM.
- [28] Christakis, M. and Bird, C. (2016). What developers want and need from program analysis: an empirical study. In D. Lo, S. Apel, and S. Khurshid, editors, *Proceedings of the 31st IEEE ACM International Conference on Automated Software Engineering*, pages 332–343, Singapore. ACM.
- [29] Christopher, T. W. (1984). Reference count garbage collection. *Software: Practice and Experience*, **14**(6), 503–507.
- [30] Collins, G. E. (1960). A method for overlapping and erasure of lists. *Communications of the ACM*, **3**(12), 655–657.
- [31] Cooper, K. D., Harvey, T. J., and Kennedy, K. (2004). Iterative data-flow analysis, revisited. Technical Report TR04-100, Rice University, Houston, Texas, USA.
- [32] Cousot, P. (1977). Asynchronous iterative methods for solving a fixed point system of monotone equations in a complete lattice. Research report 88, Laboratoire IMAG, Université scientifique et médicale de Grenoble, Grenoble, France.
- [33] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA. ACM.
- [34] Cousot, P. and Cousot, R. (1992). Abstract interpretation frameworks. *J. Log. Comput.*, **2**(4), 511–547.
- [35] Cousot, P. and Cousot, R. (2002). Modular static program analysis. In R. N. Horspool, editor, *Proceedings of the 11th International Conference on Compiler*

Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178, Grenoble, France. Springer.

- [36] Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In A. V. Aho, S. N. Zilles, and T. G. Szymanski, editors, *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, USA. ACM Press.
- [37] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The ASTRÉE analyzer. In *Proceedings of the 14th European Symposium on Programming Languages and Systems, Held as Part of the Joint European Conferences on Theory and Practice of Software*, pages 21–30, Edinburgh, United Kingdom.
- [38] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. (2009). Why does Astrée scale up? *Formal Methods in System Design*, **35**(3), 229–264.
- [39] Darais, D., Might, M., and Van Horn, D. (2015). Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In J. Aldrich and P. Eugster, editors, *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 552–571, Pittsburgh, Pennsylvania, USA. ACM.
- [40] Darais, D., Labich, N., Nguyen, P. C., and Van Horn, D. (2017). Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, **1**(ICFP), 12:1–12:25.
- [41] Deutsch, L. P. and Bobrow, D. G. (1976). An efficient, incremental, automatic garbage collector. *Communications of the ACM*, **19**(9), 522–526.
- [42] Dewey, K., Kashyap, V., and Hardekopf, B. (2015). A parallel abstract interpreter for JavaScript. In *Proceedings of the 13th Annual IEEE ACM International Symposium on Code Generation and Optimization*, pages 34–45, San Francisco, California, USA. ACM.
- [43] Distefano, D., Fähndrich, M., Logozzo, F., and O’Hearn, P. W. (2019). Scaling static analyses at Facebook. *Communications of the ACM*, **62**(8), 62–70.
- [44] Earl, C., Might, M., and Van Horn, D. (2010). Pushdown control-flow analysis of higher-order programs. In *Proceedings of the 11th Workshop on Scheme and Functional Programming*, pages 24–35, Montréal, Québec, Canada.

- [45] Earl, C., Sergey, I., Might, M., and Van Horn, D. (2012). Introspective pushdown analysis of higher-order programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, Copenhagen, Denmark.
- [46] Edvinsson, M., Lundberg, J., and Löwe, W. (2011). Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance Embedded Architectures and Compilers*, pages 45–54, Heraklion, Crete, Greece.
- [47] Fecht, C. and Seidl, H. (1999). A faster solver for general systems of equations. *Science of Computer Programming*, **35**(2), 137–161.
- [48] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. (1993). The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, USA.
- [49] Gabriel, R. P. (1985). *Performance and evaluation of LISP systems*. MIT press, Cambridge, Massachusetts, USA.
- [50] Galil, Z. and Italiano, G. F. (1991). Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, **23**(3), 319–344.
- [51] Germane, K. (2023). m-CFA exhibits perfect stack precision. In C. Hur, editor, *Proceedings of the 21st Asian Symposium on Programming Languages and Systems*, volume 14405 of *Lecture Notes in Computer Science*, pages 290–309, Taipei, Taiwan. Springer.
- [52] Germane, K. (2024). Full control-flow sensitivity for definitional interpreters. In R. Giacobazzi and A. Gorla, editors, *Proceedings of the 31st International Static Analysis Symposium*, volume 14995 of *Lecture Notes in Computer Science*, pages 120–146, Pasadena, California, USA. Springer.
- [53] Germane, K. and Adams, M. D. (2020). Liberate abstract garbage collection from the stack by decomposing the heap. In P. Müller, editor, *Proceedings of the 29th European Symposium on Programming Languages and Systems, Held as Part of the European Joint Conferences on Theory and Practice of Software*, volume 12075 of *Lecture Notes in Computer Science*, pages 197–223, Dublin, Ireland. Springer.
- [54] Germane, K. and McCarthy, J. (2021). Newly-single and loving it: improving higher-order must-alias analysis with heap fragments. *Proceedings of the ACM on Programming Languages*, **5**(ICFP), 1–28.

- [55] Gilray, T., Adams, M. D., and Might, M. (2016a). Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 407–420, Nara, Japan.
- [56] Gilray, T., Lyde, S., Adams, M. D., Might, M., and Van Horn, D. (2016b). Pushdown control-flow analysis for free. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 691–704, St. Petersburg, Florida, USA. ACM.
- [57] Glaze, D. and Van Horn, D. (2014). Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, pages 11–22, Portland, Oregon, USA.
- [58] Glaze, D., Labich, N., Might, M., and Van Horn, D. (2013). Optimizing abstract abstract machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 443–454, Boston, Massachusetts, USA.
- [59] Glaze, D., Sergey, I., Earl, C., Might, M., and Van Horn, D. (2014). Push-down flow analysis with abstract garbage collection. *Journal of Functional Programming*, **24**(2-3), 218–283.
- [60] Gotsman, A., Berdine, J., and Cook, B. (2006). Interprocedural shape analysis with separated heap abstractions. In K. Yi, editor, *Proceedings of the 13th International Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260, Seoul, Korea. Springer.
- [61] Haeupler, B., Kavitha, T., Mathew, R., Sen, S., and Tarjan, R. E. (2012). Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms*, **8**(1), 3:1–3:33.
- [62] Hardekopf, B. and Lin, C. (2011). Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th International Symposium on Code Generation and Optimization*, pages 289–298, Chamonix, France. IEEE Computer Society.
- [63] Hardekopf, B., Wiedermann, B., Churchill, B. R., and Kashyap, V. (2014). Widening for control-flow. In K. L. McMillan and X. Rival, editors, *Proceedings of the 15th International Conference Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *Lecture Notes in Computer Science*, pages 472–491, San Diego, California, USA. Springer.

- [64] Hecht, M. S. and Ullman, J. D. (1973). Analysis of a simple algorithm for global flow problems. In P. C. Fischer and J. D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 207–217, Boston, Massachusetts, USA. ACM Press.
- [65] Hecking-Harbusch, J., Quante, J., and Schlund, M. (2024). Formal runtime error detection during development in the automotive industry. In R. Dimitrova, O. Lahav, and S. Wolff, editors, *Proceedings of the 25th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 14499 of *Lecture Notes in Computer Science*, pages 3–26, London, United Kingdom. Springer.
- [66] Hind, M. and Pioli, A. (1998). Assessing the effects of flow-sensitivity on pointer alias analyses. In G. Levi, editor, *Proceedings of the 5th International Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 57–81, Pisa, Italy. Springer.
- [67] Huang, D., Qing, Y., Shang, W., Cui, H., and Zhang, J. (2024). EffiBench: Benchmarking the efficiency of automatically generated code. In A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, editors, *Proceedings of Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems*, Vancouver, BC, Canada.
- [68] Hudak, P. (1986). A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 351–363. ACM.
- [69] Hudak, P. and Bloss, A. G. (1985). The aggregate update problem in functional programming systems. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 300–314, New Orleans, Louisiana, USA.
- [70] Ishtiaq, S. S. and O’Hearn, P. W. (2001). BI as an assertion language for mutable data structures. In C. Hankin and D. Schmidt, editors, *Conference Record of The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 14–26, London, United Kingdom. ACM.
- [71] Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. K. (1998). Single and loving it: Must-alias analysis for higher-order languages. In D. B. MacQueen and L. Cardelli, editors, *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341, San Diego, California, USA. ACM.

- [72] Jeannet, B. and Miné, A. (2009). Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *Proceedings on the 21st International Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667, Grenoble, France. Springer.
- [73] Jensen, S. H., Møller, A., and Thiemann, P. (2009). Type analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium*, pages 238–255, Los Angeles, California, USA.
- [74] Jensen, S. H., Møller, A., and Thiemann, P. (2010). Interprocedural analysis with lazy propagation. In R. Cousot and M. Martel, editors, *Proceedings of the 17th International Static Analysis Symposium*, volume 6337 of *Lecture Notes in Computer Science*, pages 320–339, Perpignan, France. Springer.
- [75] Jesse, K., Ahmed, T., Devanbu, P. T., and Morgan, E. (2023). Large language models and simple, stupid bugs. In *Proceedings of the 20th IEEE ACM International Conference on Mining Software Repositories*, pages 563–575, Melbourne, Australia. IEEE.
- [76] Johnson, B., Song, Y., Murphy-Hill, E. R., and Bowdidge, R. W. (2013). Why don’t software developers use static analysis tools to find bugs? In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *Proceedings of the 35th International Conference on Software Engineering*, pages 672–681, San Francisco, California, USA. IEEE Computer Society.
- [77] Johnson, J. I. (2015). *Automating abstract interpretation of abstract machines*. Ph.D. thesis, Northeastern University.
- [78] Jones, R., Hosking, A., and Moss, E. (2016). *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC.
- [79] Jones, S. B. and Le Métayer, D. (1989). Compile-time garbage collection by sharing analysis. In *Proceedings of the 4th international conference on functional programming languages and computer architecture*, pages 54–74. ACM.
- [80] Kaestner, D., Schmidt, B., Schlund, M., Mauborgne, L., Wilhelm, S., and Ferdinand, C. (2019). Analyze this! Sound static analysis for integration verification of large-scale automotive software. In *WCX SAE World Congress Experience*. SAE Technical Paper.
- [81] Kam, J. B. and Ullman, J. D. (1977). Monotone data flow analysis frameworks. *Acta Informatica*, **7**, 305–317.

- [82] Kästner, D., Wilhelm, R., and Ferdinand, C. (2023). Abstract interpretation in industry – experience and lessons learned. In M. V. Hermenegildo and J. F. Morales, editors, *Proceedings of the 30th International Static Analysis Symposium*, volume 14284 of *Lecture Notes in Computer Science*, pages 10–27, Cascais, Portugal. Springer.
- [83] Keidel, S., Poulsen, C. B., and Erdweg, S. (2018). Compositional soundness proofs of abstract interpreters. *Proceedings of the ACM on Programming Languages*, **2**(ICFP), 72:1–72:26.
- [84] Keidel, S., Erdweg, S., and Hombücher, T. (2023). Combinator-based fixpoint algorithms for big-step abstract interpreters. *Proceedings of the ACM on Programming Languages*, **7**(ICFP), 955–981.
- [85] Kildall, G. A. (1973). A unified approach to global program optimization. In P. C. Fischer and J. D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, USA. ACM Press.
- [86] Kim, S. K., Venet, A. J., and Thakur, A. V. (2020). Deterministic parallel fixpoint computation. *Proceedings of the ACM on Programming Languages*, **4**(POPL), 14:1–14:33.
- [87] Kleene, S. C. (1952). Introduction to metamathematics.
- [88] Knauel, E. J. (2008). *A flow analysis framework for realistic Scheme programs*. Ph.D. thesis, University of Tübingen, Germany.
- [89] Kocal, A. R., Schwarz, M., Saan, S., and Seidl, H. (2026). Same engine, multiple gears: Parallelizing fixpoint iteration at different granularities. In S. Junges and G. Katz, editors, *Proceedings of the 32nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the International Joint Conferences on Theory and Practice of Software*, volume 16506 of *Lecture Notes in Computer Science*, pages 173–194, Turin, Italy. Springer.
- [90] Kramer, R., Gupta, R., and Soffa, M. L. (1994). The combining DAG: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, **5**(8), 805–813.
- [91] Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. (2018). CrySL: An extensible approach to validating the correct usage of cryptographic APIs. In

- T. D. Millstein, editor, *Proceedings of the 32nd European Conference on Object-Oriented Programming*, volume 109 of *LIPICs*, pages 10:1–10:27, Amsterdam, The Netherlands. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- [92] Le Charlier, B. and Van Hentenryck, P. (1992). A universal top-down fixpoint algorithm. Technical Report CS-92-25, CS Department, Brown University.
- [93] Lee, Y. and Ryder, B. G. (1992). A comprehensive approach to parallel data flow analysis. In *Proceedings of the 6th international conference on Supercomputing*, pages 236–247, Washington, DC, USA.
- [94] Levanoni, Y. and Petrank, E. (2006). An on-the-fly reference-counting garbage collector for Java. *ACM Transactions on Programming Language Systems*, **28**(1), 1–69.
- [95] Liang, J. T., Yang, C., and Myers, B. A. (2024). A large-scale survey on the usability of AI programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE ACM International Conference on Software Engineering*, pages 52:1–52:13, Lisbon, Portugal. ACM.
- [96] Lieberman, H. and Hewitt, C. (1983). A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, **26**(6), 419–429.
- [97] Marron, M., Hermenegildo, M. V., Kapur, D., and Stefanovic, D. (2008). Efficient context-sensitive shape analysis with graph based heap models. In L. J. Hendren, editor, *Proceedings of the 17th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software*, volume 4959 of *Lecture Notes in Computer Science*, pages 245–259, Budapest, Hungary. Springer.
- [98] Martínez, A. D., Wachenchauer, R., and Lins, R. D. (1990). Cyclic reference counting with local mark-scan. *Information Processing Letters*, **34**(1), 31–35.
- [99] Mauborgne, L. and Rival, X. (2005). Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Symposium on Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20, Edinburgh, United Kingdom. Springer.
- [100] McPeak, S., Gros, C., and Ramanathan, M. K. (2013). Scalable and incremental software bug detection. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 554–564, Saint Petersburg, Russia.

- [101] Méndez-Lojo, M., Mathew, A., and Pingali, K. (2010). Parallel inclusion-based points-to analysis. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 428–443, Reno/Tahoe, Nevada, USA.
- [102] Méndez-Lojo, M., Burtscher, M., and Pingali, K. (2012). A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–116, New Orleans, Louisiana, USA.
- [103] Might, M. (2007a). *Environment Analysis of Higher-Order Languages*. Ph.D. thesis, Georgia Institute of Technology, Atlanta, Georgia, USA.
- [104] Might, M. (2007b). Logic-flow analysis of higher-order programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 185–198, Nice, France.
- [105] Might, M. (2010). Abstract interpreters for free. In *Proceedings of the 17th International Static Analysis Symposium*, pages 407–421, Perpignan, France.
- [106] Might, M. and Shivers, O. (2006). Improving flow analyses via Γ CFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, pages 13–25, Portland, Oregon, USA.
- [107] Might, M. and Shivers, O. (2008). Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, **18**(5-6), 821–864.
- [108] Might, M., Chambers, B., and Shivers, O. (2007). Model checking via Γ CFA. In B. Cook and A. Podelski, editors, *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 59–73, Nice, France. Springer.
- [109] Might, M., Smaragdakis, Y., and Van Horn, D. (2010). Resolving and exploiting the k -CFA paradox: illuminating functional vs. object-oriented program analysis. In B. G. Zorn and A. Aiken, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–315, Toronto, Ontario, Canada. ACM.
- [110] Miné, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, **19**(1), 31–100.

- [111] Monat, R. (2021). *Static type and value analysis by abstract interpretation of Python programs with native C libraries. (Analyse statique, de type et de valeur, par interprétation abstraite, de programmes Python utilisant des librairies C)*. Ph.D. thesis, Sorbonne University, Paris, France.
- [112] Monat, R., Ouadjaout, A., and Miné, A. (2020). Value and allocation sensitivity in static Python analyses. In P. Krishnan and C. Reichenbach, editors, *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pages 8–13, London, United Kingdom. ACM.
- [113] Monat, R., Ouadjaout, A., and Miné, A. (2025). Mopsa-C with trace partitioning and autosuggestions (competition contribution). In A. Gurfinkel and M. Heule, editors, *Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Held as Part of the International Joint Conferences on Theory and Practice of Software*, volume 15698 of *Lecture Notes in Computer Science*, pages 229–235, Hamilton, Ontario, Canada. Springer.
- [114] Monniaux, D. (2005). The parallel implementation of the Astrée static analyzer. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, pages 86–96, Tsukuba, Japan.
- [115] Muth, R. and Debray, S. K. (2000). On the complexity of flow-sensitive dataflow analyses. In M. N. Wegman and T. W. Reps, editors, *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, Boston, Massachusetts, USA. ACM.
- [116] Mycroft, A. (1982). *Abstract interpretation and optimising transformations for applicative programs*. Ph.D. thesis, University of Edinburgh, United Kingdom.
- [117] Nazzal, M., Khalil, I., Khreishah, A., and Phan, N. (2024). PromSec: Prompt optimization for secure generation of functional source code with large language models (LLMs). In B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, editors, *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2266–2280, Salt Lake City, Utah, USA. ACM.
- [118] Nicolay, J., Noguera, C., De Roover, C., and De Meuter, W. (2015). Detecting function purity in JavaScript. In *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 101–110, Bremen, Germany.

- [119] Nicolay, J., Stiévenart, Q., De Meuter, W., and De Roover, C. (2019). Effect-driven flow analysis. In C. Enea and R. Piskac, editors, *Proceedings of the 20th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 11388 of *Lecture Notes in Computer Science*, pages 247–274, Cascais, Portugal. Springer.
- [120] Nielsen, L. R. (2001). A selective CPS transformation. In S. D. Brookes and M. W. Mislove, editors, *Proceedings of the 17th Conference on the Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 311–331, Aarhus, Denmark. Elsevier.
- [121] Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of program analysis*. Springer.
- [122] Oh, H. and Yi, K. (2013). Access-based abstract memory localization in static analysis. *Science of Computer Programming*, **78**(9), 1701–1727.
- [123] Oh, H., Heo, K., Lee, W., Lee, W., and Yi, K. (2012). Design and implementation of sparse global analyses for C-like languages. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 229–238. ACM.
- [124] Oh, H., Yang, H., and Yi, K. (2015). Learning a strategy for adapting a program analysis via Bayesian optimisation. In J. Aldrich and P. Eugster, editors, *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 572–588, Pittsburgh, Pennsylvania, USA. ACM.
- [125] O’Hearn, P. W., Reynolds, J. C., and Yang, H. (2001). Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Proceedings of the 15th International Workshop on Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19, Paris, France. Springer.
- [126] Ourghanlian, A. (2015). Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology*, **47**(2), 212–218. Special Issue on ISOFIC/ISSNP2014.
- [127] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2025). Asleep at the keyboard? Assessing the security of GitHub Copilot’s code contributions. *Communications of the ACM*, **68**(2), 96–105.

- [128] Perry, N., Srivastava, M., Kumar, D., and Boneh, D. (2023). Do users write more insecure code with AI assistants? In W. Meng, C. Damsgaard Jensen, C. Cremers, and E. Kirda, editors, *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 2785–2799, Copenhagen, Denmark. ACM.
- [129] Prabhu, T., Ramalingam, S., Might, M., and Hall, M. W. (2011). EigenCFA: accelerating flow analysis with GPUs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 511–522, Austin, Texas, USA.
- [130] Qiu, R., Zeng, W. W., Ezick, J., Lott, C., and Tong, H. (2025). How efficient is LLM-generated code? A rigorous & high-standard benchmark. In *Proceedings of the 13th International Conference on Learning Representations*, Singapore.
- [131] Reps, T. W., Horwitz, S., and Sagiv, S. (1995). Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, USA.
- [132] Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. In J. J. Donovan and R. Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM.
- [133] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark. IEEE Computer Society.
- [134] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, **74**(2), 358–366.
- [135] Rinetzký, N., Bauer, J., Reps, T. W., Sagiv, S., and Wilhelm, R. (2005). A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 296–309, Long Beach, California, USA.
- [136] Roditty, L. and Zwick, U. (2008). Improved dynamic reachability algorithms for directed graphs. *SIAM Journal on Computing*, **37**(5), 1455–1471.
- [137] Rodriguez, J. and Lhoták, O. (2011). Actor-based parallel dataflow analysis. In *Proceedings of the 20th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software*, pages 179–197, Saarbrücken, Germany.

- [138] Roy, R., S., S., and Paleri, V. (2024). Partial redundancy elimination in two iterative data flow analyses. In J. Aldrich and G. Salvaneschi, editors, *Proceedings of the 38th European Conference on Object-Oriented Programming*, volume 313 of *LIPICs*, pages 35:1–35:19, Vienna, Austria. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- [139] Sabry, A. and Felleisen, M. (1994). Is continuation-passing useful for data flow analysis? In V. Sarkar, B. G. Ryder, and M. L. Soffa, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Orlando, Florida, USA. ACM.
- [140] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. (2018). Lessons from building static analysis tools at Google. *Communications of the ACM*, **61**(4), 58–66.
- [141] Schmidt, D. A. (1995). Natural-semantics-based abstract interpretation (preliminary version). In A. Mycroft, editor, *Proceedings of the 2nd International Static Analysis Symposium*, volume 983 of *Lecture Notes in Computer Science*, pages 1–18, Glasgow, United Kingdom. Springer.
- [142] Schmidt, D. A. (1997). Abstract interpretation in the operational semantics hierarchy. *BRICS Report Series*, (2).
- [143] Sergey, I., Devriese, D., Might, M., Midtgaard, J., Darais, D., Clarke, D., and Piessens, F. (2013). Monadic abstract interpreters. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 399–410, Seattle, Washington, USA.
- [144] Shahriyar, R., Blackburn, S. M., and Frampton, D. (2012). Down for the count? Getting reference counting back in the ring. In *Proceedings of the International Symposium on Memory Management*, pages 73–84, Beijing, China.
- [145] Shavit, N. and Touitou, D. (1995). Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, Ottawa, Ontario, Canada. Association for Computing Machinery.
- [146] Shivers, O. (1988). Control-flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, Atlanta, Georgia, USA.
- [147] Shivers, O. (1991). *Control-flow analysis of higher-order languages*. Ph.D. thesis, Carnegie Mellon University.

- [148] Sigurdhsson, R. L. (2016). *Practical performance of incremental topological sorting and cycle detection algorithms*. Master’s thesis, Chalmers University of Technology.
- [149] Sobb, T. M. and Turnbull, B. P. (2025). CrowdStrike: Lessons in mission assurance to build resilience for complex cyber-physical-social systems. *IEEE Access*, **13**, 205324–205337.
- [150] Souyris, J., Wiels, V., Delmas, D., and Delseny, H. (2009). Formal verification of avionics software products. In A. Cavalcanti and D. Dams, editors, *Proceedings of the 2nd World Congress on Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 532–546, Eindhoven, The Netherlands. Springer.
- [151] Stiévenart, Q., Nicolay, J., De Meuter, W., and De Roover, C. (2016a). Building a modular static analysis framework in Scala (tool paper). In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala*, pages 105–109, Amsterdam, The Netherlands.
- [152] Stiévenart, Q., Vandercammen, M., De Meuter, W., and De Roover, C. (2016b). Scala-AM: A modular static analysis framework. In *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 85–90, Raleigh, North Carolina, USA.
- [153] Stiévenart, Q., Nicolay, J., De Meuter, W., and De Roover, C. (2019). A general method for rendering static analyses for diverse concurrency models modular. *Journal of Systems and Software*, **147**, 17–45.
- [154] Stiévenart, Q., Van Es, N., Van der Plas, J., and De Roover, C. (2021). A parallel worklist algorithm and its exploration heuristics for static modular analyses. *Journal of Systems and Software*, **181**, 111042.
- [155] Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, **1**(2), 146–160.
- [156] Tarjan, R. E. (1985). Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, **6**(2), 306–318.
- [157] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, **5**(2), 285–309.
- [158] Urban, C. (2020). What programs want. *arXiv*, **abs/2007.10688**.

- [159] Vaithilingam, P., Zhang, T., and Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In S. D. J. Barbosa, C. Lampe, C. Appert, and D. A. Shamma, editors, *Extended Abstracts of the Conference on Human Factors in Computing Systems*, pages 332:1–332:7, New Orleans, Louisiana, USA. ACM.
- [160] Van der Plas, J., Stiévenart, Q., Van Es, N., and De Roover, C. (2020). Incremental flow analysis through computational dependency reification. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–36, Adelaide, Australia. IEEE.
- [161] Van Es, N., Nicolay, J., Stiévenart, Q., D’Hondt, T., and De Roover, C. (2016). A performant Scheme interpreter in asm.js. In S. Ossowski, editor, *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1944–1951, Pisa, Italy. ACM.
- [162] Van Es, N., Stiévenart, Q., Nicolay, J., D’Hondt, T., and De Roover, C. (2017a). Implementing a performant Scheme interpreter for the web in asm.js. *Computer Languages, Systems & Structures*, **49**, 62–81.
- [163] Van Es, N., Vandercammen, M., and De Roover, C. (2017b). Incrementalizing abstract interpretation. In S. Demeyer, A. Parsai, G. Laghari, and B. van Bladel, editors, *Proceedings of the 16th edition of the BELgian-NEtherlands software eVOLution symposium*, CEUR Workshop Proceedings, pages 31–35, Antwerp, Belgium. CEUR-WS.org.
- [164] Van Es, N., Stiévenart, Q., and De Roover, C. (2019). Garbage-free abstract interpretation through abstract reference counting. In A. F. Donaldson, editor, *Proceedings of the 33rd European Conference on Object-Oriented Programming*, LIPICs, pages 10:1–10:33, London, United Kingdom. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.
- [165] Van Es, N., Van der Plas, J., Stiévenart, Q., and De Roover, C. (2020a). MAF: A framework for modular static analysis of higher-order languages. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 37–42, Adelaide, Australia. IEEE.
- [166] Van Es, N., Stiévenart, Q., Van der Plas, J., and De Roover, C. (2020b). A parallel worklist algorithm for modular analyses. In *Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 1–12, Adelaide, Australia. IEEE.

- [167] Van Es, N., Vandenbogaerde, B., and De Roover, C. (2025). Delta store semantics: Abstract garbage collection for abstract definitional interpreters. In H. Oh and Y. Sui, editors, *Proceedings of the 32nd International Static Analysis Symposium*, Lecture Notes in Computer Science, pages 386–421, Singapore. Springer.
- [168] Van Horn, D. and Might, M. (2010). Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 51–62, Baltimore, Maryland, USA.
- [169] Vandenbogaerde, B., Verbelen, S., Van Es, N., and De Roover, C. (2025). Monarch: A modular framework for abstract definitional interpreters in Haskell. In H. Oh and Y. Sui, editors, *Proceedings of the 32nd International Static Analysis Symposium*, Lecture Notes in Computer Science, pages 357–385, Singapore. Springer.
- [170] Vardoulakis, D. (2012). *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. Ph.D. thesis, Northeastern University.
- [171] Vardoulakis, D. and Shivers, O. (2011). CFA2: a context-free approach to control-flow analysis. *Logical Methods in Computer Science*, **7**(2).
- [172] Venet, A. and Brat, G. P. (2004). Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 231–242, Washington, DC, USA.
- [173] Verbelen, S., Vandenbogaerde, B., Van der Plas, J., Van Es, N., and De Roover, C. (2024). Abstract slicing to improve the speed of static program analysis. In G. Perrouin, B. Vanderose, and X. Devroey, editors, *Proceedings of the 23rd Belgium-Netherlands Software Evolution Workshop*, CEUR Workshop Proceedings, pages 134–145, Namur, Belgium. CEUR-WS.org.
- [174] Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In J. Ferrante and K. S. McKinley, editors, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–41, San Diego, California, USA. ACM.
- [175] Weeks, S., Jagannathan, S., and Philbin, J. (1994). A concurrent abstract interpreter. *LISP and Symbolic Computation*, **7**(2-3), 173–193.

- [176] Wegman, M. N. and Zadeck, F. K. (1991). Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, **13**(2), 181–210.
- [177] Wei, G., Decker, J. M., and Rompf, T. (2018). Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, **2**(ICFP), 105:1–105:28.
- [178] Weiser, M. D. (1984). Program slicing. *IEEE Transactions on Software Engineering*, **10**(4), 352–357.
- [179] Weizenbaum, J. (1969). Recovery of reentrant list structures in SLIP. *Communications of the ACM*, **12**(7), 370–372.
- [180] Whiting, T. and Germane, K. (2026). HMCFA: A precise and practical big-step control flow analysis for effect handlers. *Proc. ACM Program. Lang.*, **10**(ICFP), 892–920.
- [181] Xie, Y. and Aiken, A. (2007). Saturn: A scalable framework for error detection using Boolean satisfiability. *ACM Transactions on Programming Languages and Systems*, **29**(3), 16.
- [182] Yang, H. and O’Hearn, P. W. (2002). A semantic basis for local reasoning. In M. Nielsen and U. Engberg, editors, *Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures, Held as Part of the Joint European Conferences on Theory and Practice of Software*, volume 2303 of *Lecture Notes in Computer Science*, pages 402–416, Grenoble, France. Springer.