Downloaded from orbit.dtu.dk on: Apr 28, 2024

DTU Library

An algebraic approach to analysis of recursive and concurrent programs

Terepeta, Michal Tomasz

Publication date:
2013

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Terepeta, M. T. (2013). *An algebraic approach to analysis of recursive and concurrent programs*. Technical University of Denmark. IMM-PHD-2013 No. 307

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

An algebraic approach to analysis of recursive and concurrent programs

Michał Terepeta

Kongens Lyngby 2013
IMM-PHD-2013-307

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 303 B, DK-2800 Kgs. Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

IMM-PHD: ISSN 0909-3192, ISBN 0909-3192

Summary

This thesis focuses on formal techniques based on static program analysis, model checking and abstract interpretation that offer means for reasoning about software, verification of its properties and discovering potential bugs.

First, we investigate an algebraic approach to static analysis and explore its connections to abstract interpretation framework. We introduce the notion of a flow algebra, which is an algebraic structure similar to semirings, but closer to the classical monotone frameworks. We also generalize Galois connections to flow algebras and discuss when a flow algebra is an upper-approximation of (or induced from) another flow algebra.

Furthermore, we show how flow algebras can be used in communicating or weighted pushdown systems. To achieve that, we show that it is possible to relax some of the requirements imposed by original formulation of those techniques without compromising the soundness or completeness results.

Moreover, we present a new application of pushdown systems in the context of an aspect-oriented process calculus. The addition of aspect-oriented features makes it possible for a process to exhibit a recursive structure. We show how one can faithfully model and analyze such a language.

We also introduce an abstract domain that symbolically represents the messages sent between the concurrently executing processes. It stores prefixes or suffixes of communication traces including various constraints imposed on the messages. Since the problem has exponential complexity, we also present a compact data structure as well as efficient algorithms for the semiring operations.

Apart from that, we discuss an improvement to Pre^* and $Post^*$ algorithms for pushdown systems, making it possible to directly use program representations such as program graphs. We present a modular library implementing those algorithms, which also provides a lot of flexibility with respect to, e.g., various constraints solvers.

Finally, we describe one such experimental solver based on Newton's method. It allows solving equation systems over abstract domains that were not accommodated by other solving techniques, e.g., Kleene iteration. We present such a domain and provide a preliminary evaluation of our implementation.

To conclude, we believe the thesis presents a number of contributions interesting both from the theoretical point of view as well as from an implementation one.

Resumé

Afhandlingen fokuserer på formelle teknikker, baseret på statiske program analyse, model tjek og abstrakt fortolkning. Disse teknikker kan bruges til at vurdere software, verificere visse egenskaber ved software, og finde potentielle fejl.

Først undersøger vi en algebraisk tilgang til statisk analyse og udforsker dens forbindelser til abstrakt fortolkning. Vi introducerer begrebet flow algebra, der er en algebraisk struktur svarende til semiringe, men som er mere beslægtet med de klassiske monotone frameworks. Vi viser hvordan man kan generalisere Galois-forbindelser til flow algebraer og diskuterer hvornår en flow algebra er en over-approksimation af (eller er induceret fra) en anden flow algebra.

Dernæst viser vi, hvordan flow algebraer kan anvendes i kommunikerende eller vægtede push-down systemer. For at opnå dette, viser vi, at det er muligt at slække på nogle af de krav pålagt af den oprindelige formulering af disse teknikker uden at det går ud over de ønskede sundheds og fuldstændigheds resultater.

Desuden præsenterer vi en ny anvendelse af push-down systemer i forbindelse med et aspekt-orienteret proces sprog. Tilføjelsen af aspekt-orienterede kendetegn gør det muligt for de processer at fremvise en rekursiv struktur. Vi viser, hvordan man kan modellere og analysere et sådant sprog.

Vi også indfører et abstrakt domæne, som symbolsk repræsenterer de beskeder der sendes mellem de samtidigt udførende processer. Det benyttes til at huske præfikser eller suffikser af kommunikations forløb, og inkluderer også betingelser som påtrykkes af beskeder der udveksles. Da problemets kompleksitet er eksponentiel, har vi også præsenteret en kompakt datastruktur med samt effektive

algoritmer til semiring operationer.

Ydermere diskuterer vi en forbedring af Pre* og Post* algoritmerne for push-down systemer, som gør det muligt direkte at bruge program repræsentationer som for eksempel program grafer. Vi præsenterer en modulær bibliotek som implementerer disse algoritmer; dermed opnås stor fleksibilitet med hensyn til forskellige metoder til løsning af constraint systemer.

Endelig beskriver vi en eksperimentel metode til løsning af constraint systemer baseret på Newtons metode. Den gør det muligt at løse ligningssystemer over abstrakte domæner, som tidligere løsningsteknikker, f.eks Kleene iteration, ikke kunne magte. Vi præsenterer en sådant domæne og giver en foreløbig evaluering af vores implementation.

Sluttelig mener vi, at afhandlingen præsenterer en mængde af bidrag som er interessante både fra teoretisk og et praktisk synspunkt.

Preface

This thesis was prepared at DTU Compute (formerly DTU Informatics), Technical University of Denmark in partial fulfillment of the requirements for acquiring a Ph.D. degree in computer science.

The Ph.D. study has been carried out under the supervision of Professor Hanne Riis Nielson and Professor Flemming Nielson in the period of August 2010 to July 2013.

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. Some of the work presented is based on articles co-authored with my supervisors (Chapters 3, 5 and 6), Piotr Filipiuk (Chapter 3) and Maximilian Schlund (Chapter 9).

Lyngby, July 2013

Michał Terepeta

Acknowledgements

This thesis would not be possible without the help of many people for which I am truly grateful.

First of all, I would really like to thank my family for their encouragement and support. They were always there for me whenever I needed them the most.

I would also like to thank my supervisors Professor Hanne Riis Nielson and Professor Flemming Nielson from whom I have learned so much. Moreover, I am happy I had the opportunity to be part of the Language Based Technology section at the Technical University of Denmark and would like to thank everyone there for all the interesting discussions and an enjoyable atmosphere.

Finally, I would also like to thank Professor Javier Esparza and all the members of the Chair for Foundations of Software Reliability and Theoretical Computer Science at the Technical University of Munich for making my research stay there so exciting and interesting.

Contents

Chapter 1

Introduction

1.1 Software verification

Modern societies depend on software in almost every aspect of every-day life, from PCs, phones or tablets to cars, medical equipment and avionics. We not only use software in more and more places, but we also increase the complexity of the tasks performed with it. In other words we want more functionality, performance, ease of use and at the same time we also expect that it is correct, robust and reliable. Those goals often work against each other — the higher the complexity, the more difficult it is to ensure correctness. Software that was considered advanced and complex a decade ago is often considered to be quite basic by today's standards. With all of this in mind it is not difficult to see why creating high quality software is a huge challenge. Therefore, many people around the world work on different ways to improve the way we create software — from the design and development processes to different ways of testing and performing code reviews. Many of those improvements are not formal in the sense of being based on mathematical rigor, yet often they are quite effective [40]. On the other hand, as observed by Dijkstra [22]:

> [...] program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Therefore, in many areas that require highest levels of reliability, development processes or testing are not always enough. Apart from that, in the last decade we have witnessed another great shift — the rise of multi-core systems. Nowadays, the performance gains are mainly due to more cores and CPUs and not as much due to faster cores themselves. This puts additional pressure on creating parallel and concurrent software in order to utilize modern hardware. And again, it only increases the complexity of the software and makes it more difficult to reason about it. It is also much harder to test it, since many errors can only be observed in a very specific conditions, e.g., only for highly unusual program interleavings.

Due to such challenges, in recent years there has been a lot of activity and progress in using formal methods to improve the software. This includes creating tools on top of formal theories that help in finding bugs or even allowing to verify various properties of the software. Moreover, they can be used at different levels of abstraction and different stages of the design and implementation of the system. The main approaches to formal analysis and verification of various systems are static analysis, abstract interpretation and model checking.

Static analysis is a technique of computing approximate behaviors of programs statically at compile-time. It was initially created and developed in the context of optimizing compilers, for the purpose of improving the performance of the generated code. However, currently the main focus of static analysis is shifting towards the problem of bug finding and software verification. One of the main strengths of static analysis is that it allows to statically prove certain properties of programs. Thus, they will hold for every execution of the program, for all possible input value, etc. Of course, the price for this is that the analyses can often be quite complex and computationally demanding, since they have to consider all those various possibilities.

Another close technique is abstract interpretation [19], which provides a rigorous approach to creating sound analyses. This is usually achieved by establishing a connection (more precisely a Galois connection) between the static semantics of a program and some abstract domain. By formally describing the relationship of an analysis to the semantics of the program one can create analyses that are semantically correct by construction. Furthermore, it provides a framework for creating new abstract domains from already existing ones [20].

Finally, model checking [3] is an automated technique for determining whether a given model meets a given specification. This is achieved by exhaustive exploration of the model, which means that it is possible to determine whether the system satisfies the specification. Moreover, with modern, sophisticated algorithms and data structures it is possible to model check very large systems (e.g., the technique is commonly used for hardware verification). However, due

to the state space explosion problem, it is often not directly applicable to software verification. There are many different formalism in which it is possible to define the specification, such as LTL, CTL, PCTL, etc.

Unfortunately most of the interesting questions about software are undecidable [68]. Therefore, if the analysis is to be sound (i.e., admit only false-positives) then some over-approximation is often necessary. Alternatively some approaches go for under-approximation, which can be quite effective in bug finding. However, they obviously cannot be used for verification, since they admit false-negatives, i.e., the analysis might not find any problems, even though some exist. Therefore, nowadays when analyzing and verifying complex software systems, one often uses a combination of all of the above approaches. And so the popular term "software model checking" [44] is a bit of a misnomer and usually includes techniques from all of these areas.

1.2 Contributions

The contributions of this thesis apply several slightly different areas. However, they are all connected and can be expressed in a single sentence, namely the main theme of this thesis is:

> An algebraic approach to analysis and verification of recursive and concurrent systems.

This presents some of the main topics that we focus in this thesis: an *algebraic approach* to program analysis, analysis/verification of programs containing *recursive* structure and which are *concurrent*. We will briefly discuss those three points one by one.

Structures such as monoids and semirings [23] have already been used in many contexts of computer science. Semirings in particular appear to be quite useful for program analysis purposes [66, 67, 10, 11], where the summation operator corresponds to merging of the information from different paths and the multiplication operator to concatenating the paths. Moreover, many analyses can be expressed in terms of semirings. However, there are many classical analyses that do not exactly fit into this formulation. One of the contributions of the thesis is to introduce flow algebras [30], that are less restrictive and admit more analyses. We study the how they can be used in the context of abstract interpretation in Chapter 3, where we generalize Galois connections from complete lattices to flow algebras. Acknowledgement: Chapter 3 is based on [30].

Furthermore, we also investigate using flow algebras for interprocedural analysis. To that end we investigate the communicating/weighted pushdown systems that use semirings to represent the analysis information. In Chapter 5 we establish that it is possible to use flow algebras instead, thus disposing of some of the strong requirements imposed by the semirings on the abstract domains. Acknowledgement: Chapter 5 is based on [77].

We continue exploring algebraic approach to static analysis in Chapter 9, where we experiment with an implementation of Newton's method generalized to ω-continuous semirings. The technique allows us to compute the least solution of a system of equations in some cases when Kleene iteration is not able to. Acknowledgement: Chapter 9 is based on [71].

Another topic that this thesis is exploring is the concurrency based on channel communication. The main starting point is the concept of communicating pushdown systems, where the language of synchronization actions of different processes is over-approximated and then used in order to prove unreachability of error states. We use the main ideas of the approach in Chapter 6 and propose a novel analysis approach to a process calculi with aspect-oriented features. It turns out that introducing aspects can introduce a recursive structure to such processes making the analysis much more difficult. In order to handle this case we use pushdown systems in quite a different way than usual, which allows us to achieve both good precision and model the aspect-oriented features correctly. Acknowledgement: Chapter 6 is based on [76].

Motivated by some of the challenges encountered during the above work, in Chapter 7 we describe a new symbolic abstract domain that extends the ith-prefix and i^{th}-suffix abstractions of [14]. Our abstract domain represents the variables symbolically and allows to include constraints over those variables, making it possible to achieve high precision and good performance of the analysis. Furthermore, we also develop and implement a data structure to represent the communication traces in a compact and efficient way, along with algorithms for language union and concatenation (cf. flow algebra summation and multiplication operators). The satisfiability checking of the constraints is is made quite easy by modern SMT solvers.

As already mentioned above, one of the major areas of focus of this thesis is the analysis of systems with some recursive structure, e.g., procedural programs. We have decided to use pushdown systems annotated with algebraic weights to model such systems. Pushdown systems are already a popular technique for such problems. One of our main contributions is joining the work on flow algebras and weighted pushdown systems — in Chapter 5 we show by reformulating the Pre^* and $Post^*$ algorithms, it is possible to use flow algebras instead of semirings. We prove both the soundness (the analysis result is a safe approximation of join-

over-all-valid paths), as well as completeness (provided the flow algebra satisfies certain additional properties, the analysis result coincides with the join-over-all-valid paths) results. Another of our contributions in this area is showing how one can use pushdown systems for modelling an aspect-oriented process calculi, in Chapter 6.

Finally, the thesis also features contributions that are interesting from the implementation point of view. In Chapter 7 we introduce an efficient data structure for representing finite languages along with optimized algorithms for both the binary operations. We also extend it with the ability to capture symbolic constraints that makes it possible to avoid some of the problems connected to using weighted pushdown systems for message passing (and not only synchronization). Moreover, in Chapter 8 we present improved algorithms for computing Pre^* and $Post^*$ based on graph traversal. We believe that even though they do not offer an asymptotic improvement in performance, they do make it much easier to incorporate weighted pushdown systems into existing tools and with less overhead. Furthermore, by decoupling the constraint solving from the Pre^* and $Post^*$ computations we make it possible to use a dedicated solver specialized for a particular domain as well as to compute solutions of various domains in parallel or only when needed. This makes it possible to use various constraint solvers, e.g., using Kleene iteration or solvers based on the recent results in generalizing of Newton's method to ω-continuous semirings.

In conclusion, the thesis contains many contributions both from the theoretical point of view, as well as from the practical one. We believe that this is essential since software verification techniques should be based on rigorous foundations but at the same time, to be really useful, it should be possible to implement and apply them.

Chapter 2

Preliminaries

This section provides a quick introduction to some of the basic, but essential background as well as notation that will be used throughout this thesis. Most of this section follows [59], where the reader can find more detailed description along with the proofs for most of the results. However, we also introduce some important algebraic concepts that are not presented in [59].

2.1 Partial orders

We start with one of the most fundamental concepts, namely that of a partial ordering.

Definition 2.1 A partial ordering is a binary relation $\sqsubseteq$ on some set L that is:

- reflexive: $\forall l \in L : l \sqsubseteq l$
- transitive: $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \implies l_1 \sqsubseteq l_3$
- anti-symmetric: $\forall l, l' \in L : l \sqsubseteq l' \wedge l' \sqsubseteq l \implies l = l'$

Definition 2.2 A partially ordered set (poset) $(L, \sqsubseteq)$ is a set L with a partial ordering $\sqsubseteq$.

We sometimes write $\sqsubseteq_L$ if L is not clear from the context.

When we consider subsets of a poset, it is often desirable to distinguish some elements of the poset that are "larger" or "smaller" than all the elements of the subset.

Definition 2.3 An element $l_b \in L$ is an upper bound of $Y \subseteq L$ if

$$\forall l \in Y : l \sqsubseteq l_b$$

Additionally l_b is a least upper bound if for all upper bounds l of Y we have that $l_b \sqsubseteq l$.

The (greatest) lower bound is defined similarly:

Definition 2.4 An element $l_b \in L$ is a lower bound of $Y \subseteq L$ if

$$\forall l \in Y : l_b \sqsubseteq l$$

Additionally l_b is a greatest lower bound if for all lower bounds l of Y we have that $l \sqsubseteq l_b$.

It is important to emphasize that in a partially ordered set not every subset Y has a least upper bound or greatest lower bound. However, if they exist they are unique due to the anti-symmetry of $\sqsubseteq$. We denote them as $\bigsqcup Y$ (called *join*) and $\bigsqcap Y$ (called *meet*).

Definition 2.5 A subset $Y \subseteq L$ of a poset $(L, \sqsubseteq)$ is a chain if

$$\forall l_1, l_2 \in Y : l_1 \sqsubseteq l_2 \vee l_2 \sqsubseteq l_1$$

Many algorithms in static analysis expect certain properties of the domain in order to always terminate (e.g., calculating the least fixed point using Kleene iteration). One of such requirements is the *ascending chain condition* that we introduce below. An ascending chain is simply a sequence $(l_n)_{n \in \mathbb{N}}$ such that

$$\forall i \leq j : l_i \sqsubseteq l_j$$

We say that a sequence $(l_n)_n$ eventually stabilizes if and only if

$$\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N} : n_0 \leq n \implies l_{n_0} = l_n$$

Definition 2.6 A partially ordered set L satisfies the ascending chain condition if and only if all ascending chains eventually stabilize.

2.2 Lattices and functions over lattices

Definition 2.7 A complete lattice $(L, \sqsubseteq, \sqcup, \sqcap)$ is a partially ordered set such that every subset has least upper and greatest lower bounds.

Since we will often talk about various functions defined over lattices, we introduce some of the important concepts below.

Definition 2.8 A function $f : L \to L$ is monotone if and only if

$$\forall l_1, l_2 : l_1 \sqsubseteq l_2 \implies f(l_1) \sqsubseteq f(l_2)$$

Definition 2.9 A function $f : L \to L$ is distributive if and only if

$$\forall l_1, l_2 : f(l_1) \sqcup f(l_2) = f(l_1 \sqcup l_2)$$

Definition 2.10 A function $f : L \to L$ is affine if and only if

$$\forall Y \subseteq L : Y \neq \emptyset : \bigsqcup\{f(l) \mid l \in Y\} = f(\bigsqcup Y)$$

Definition 2.11 A function $f : L \to L$ is completely distributive if and only if

$$\forall Y \subseteq L : \bigsqcup\{f(l) \mid l \in Y\} = f(\bigsqcup Y)$$

Since a strict function is a function such that $f(\bot) = \bot$ we could also say that f is completely distributive if and only if it is both affine and strict.

In many practical situations it is useful to abstract from some precise but computationally expensive lattice to a less precise but more tractable one. This can be achieved by establishing a *Galois connection* and is a standard technique in abstract interpretation [19, 20].

Definition 2.12 A Galois connection is a tuple (L, α, γ, M) such that L and M are complete lattices and α, γ are monotone functions called abstraction and concretization functions. They satisfy:

$$\alpha \circ \gamma \sqsubseteq \lambda m.m$$
$$\gamma \circ \alpha \sqsupseteq \lambda l.l$$

We also distinguish a special case of Galois connections called Galois insertion.

Definition 2.13 A Galois insertion is a Galois connection (L, α, γ, M) such that

$$\alpha \circ \gamma = \lambda m.m$$

Lemma 2.14 *For a Galois connection (L, α, γ, M) the following claims are equivalent:*

- (L, α, γ, L) *is a Galois insertion*
- α *is surjective:* $\forall m \in M : \exists l \in L : \alpha(l) = m$
- γ *is injective:* $\forall m_1, m_2 \in M : \gamma(m_1) = \gamma(m_2) \implies m_1 = m_2$
- γ *is order-similarity:* $\forall m_1, m_2 \in M : \gamma(m_1) \sqsubseteq \gamma(m_2) \iff m_1 \sqsubseteq m_2$

The proof for the lemma is available in [59].

2.3 Monoids and semirings

Since both weighted and communicating pushdown systems are using semirings, we will introduce some of the basic definitions associated with them [23], starting with the definition of a monoid.

Definition 2.15 A monoid is a tuple $(M, \otimes, \bar{1})$ such that M is non-empty, $\otimes$ is an associative operator on M and $\bar{1}$ is a neutral element for $\otimes$, i.e.,

$$\forall a \in M : a \otimes \bar{1} = \bar{1} \otimes a = a$$

A monoid is *idempotent* if $\otimes$ operator is idempotent, that is

$$\forall a \in M : a \otimes a = a$$

Similarly it is *commutative* if the operator is commutative, in which case we usually use the symbol $\oplus$ to denote it (and also use $\bar{0}$ to for the neutral element):

$$\forall a, b \in M : a \oplus b = b \oplus a$$

A commutative monoid $(M, \oplus, \bar{0})$ is *naturally ordered* if the relation defined as

$$\forall a, b \in M : a \sqsubseteq b \iff \exists c \in M : a \oplus c = b$$

is a partial order. Moreover, if the monoid is idempotent then it is naturally ordered and we have that

$$\forall a, b \in M : a \sqsubseteq b \iff a \oplus b = b$$

and $\oplus$ is the least upper bound operator. Note that this corresponds to a join semi-lattice.

Now we are ready do define the semiring structure.

Definition 2.16 A semiring is a tuple $(S, \oplus, \otimes, \bar{0}, \bar{1})$ such that

- $(S, \oplus, \bar{0})$ is a commutative monoid (hence $\bar{0}$ is a neutral element for $\oplus$)
- $(S, \otimes, \bar{1})$ is a monoid (hence $\bar{1}$ is a neutral element for $\otimes$)
- $\otimes$ distributes over $\oplus$, that is

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$$
$$(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$$

- $\bar{0}$ is an annihilator for $\otimes$, that is $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$

Similarly to the above, we call a semiring idempotent if $\oplus$ is idempotent, and commutative if $\otimes$ is commutative. The ordering for idempotent semiring is defined in the same way as for idempotent and commutative monoids, with the additional requirement that $\otimes$ preserves the order (i.e., is monotonic).

Note that an idempotent semiring $(S, \oplus, \otimes, \bar{0}, \bar{1})$ is a join semilattice, where:

- $\sqcup = \oplus$
- $\perp = \bar{0}$
- $a \sqsubseteq b$ if and only if $a \oplus b = b$.

Finally, a bounded semiring is a semiring with no infinite ascending chains [66, 67].

Chapter 3

Intraprocedural analysis using flow algebras

Semirings have been used in computer science in a variety of contexts and have also found their way into the areas of static analysis and software model checking. Examples include weighted [66, 67] and communicating pushdown systems [10, 11], which use them to annotate the pushdown rules and create weighted automata. But the ideas of using monoids for static analysis can be already found in [69]. In this chapter we introduce the notion of a *flow algebra* and explore how they fit in the abstract interpretation framework. Flow algebras are algebraic structures that are less restrictive than idempotent semirings in that they replace distributivity with monotonicity and dispense with the annihilation property; therefore they are closer to the approach taken by Monotone Frameworks and other classical analyses. We present a generic framework for static analysis based on flow algebras and program graphs.[1] Furthermore, we generalize *Galois connections* from complete lattices to flow algebras. Our framework allows to *induce* new flow algebras using Galois connections such that correctness of the analyses is preserved. We believe that this development offers additional insight into the use of such algebraic structures in static analysis and abstract interpretation. The approach is illustrated for a *mutual exclusion* algorithm.

[1] Program graphs are often used in model checking to model concurrent and distributed systems.

The structure of the chapter is as follows. In Section 3.2 we introduce the flow algebras and then we show how Galois connections are defined for them in Section 3.3. We perform a similar development for program graphs by defining them in Section 3.4, presenting how to express analysis using them in Section 3.5 and then describing Galois connections for program graphs in Section 3.6. Finally, we present a motivating example of our approach in Section 3.7 and conclude in Section 3.8.

Acknowledgement: This chapter is based on joint work with Piotr Filipiuk and my supervisors, which was published as [30].

3.1 Introduction

In the classical approach to static analysis we usually use the notion of Monotone Frameworks [45, 1] that work over flow graphs as an abstract representation of a program. A Monotone Framework, primarily used for data-flow analysis [49], consists of a complete lattice describing the properties of the system (without infinite ascending chains) and transfer functions over that lattice (that are monotone). When working with complete lattices, one can take advantage of Galois connections to induce new analyses or to over-approximate them [20]. Recall that a Galois connection is a correspondence between two complete lattices that consists of abstraction and concretization functions. It is often used to move an analysis from a computationally expensive lattice to a less costly one and plays a crucial role in abstract interpretation [19].

In this chapter we introduce a similar framework that uses flow algebras to define analyses. Flow algebras are algebraic structures consisting of two monoids quite similar to idempotent semirings, which, as presented in the preliminaries, have already been used in software analysis. However, flow algebras are less restrictive and allow to directly express some of the classical analysis, which is simply not possible with idempotent semirings.[2] Furthermore, as representation of the system under consideration we use *program graphs*, in which actions label the edges rather than the nodes. The main benefit of using program graphs is that we can model concurrent systems in a very straightforward manner. Moreover, since a model of a concurrent system is also a program graph, all the results are applicable both in the sequential as well as in the concurrent setting.

We also define both the Meet Over all Paths (*MOP*) solution of the analysis

[2]Note that it is possible to introduce some additional artificial elements to the abstract domain and add special cases to the $\otimes$ operator in order to express such analyses in terms of semirings.

as well as a set of constraints that can be used to obtain the Maximal Fixed Point (*MFP*) solution. By establishing that the solutions to the constraint system constitute a Moore family, we know that there always exists a least (i.e., best) solution. Intuitively the main difference between *MOP* and *MFP* is that the former expresses what we would like to compute, whereas the latter is sometimes less accurate but computable in some cases where *MOP* is not. Finally, we establish that they coincide in case of distributive analyses.

We also extend the notion of Galois connections to flow algebras and program graphs. This allows us to easily create new analyses based on existing ones. In particular we can create Galois connections between the collecting semantics (defined in terms of our framework) and various analyses, which ensures their semantic correctness.

Finally, we apply our results to a variant of the Bakery mutual exclusion algorithm [56]. By inducing an analysis from the collecting semantics and a Galois insertion, we are able to prove the correctness of the algorithm. Thanks to our previous developments we know that the analysis is semantically correct.

3.2 Flow algebra

3.2.1 Definition

Now we will introduce the notion of a *flow algebra*.[3] It is an algebraic structure that is less restrictive than an idempotent semiring — flow algebras do not require the distributivity and annihilation properties. Instead we replace the first one with a monotonicity requirement and dispense with the second one. A flow algebra is formally defined as follows.

Definition 3.1 A flow algebra is a structure of the form $(F, \oplus, \otimes, \bar{0}, \bar{1})$ such that:

- $(F, \oplus, \bar{0})$ is an idempotent commutative monoid:
  - $(f_1 \oplus f_2) \oplus f_3 = f_1 \oplus (f_2 \oplus f_3)$
  - $\bar{0} \oplus f = f \oplus \bar{0} = f$
  - $f_1 \oplus f_2 = f_2 \oplus f_1$

[3] The name comes from the idea of performing data**flow** analyses using an **algebra**ic structure.

  - $f \oplus f = f$

- $(F, \otimes, \bar{1})$ is a monoid:
  - $(f_1 \otimes f_2) \otimes f_3 = f_1 \otimes (f_2 \otimes f_3)$
  - $\bar{1} \otimes f = f \otimes \bar{1} = f$

- $\otimes$ is monotonic in both arguments:
  - $f_1 \leq f_2 \;\Rightarrow\; f_1 \otimes f \leq f_2 \otimes f$
  - $f_1 \leq f_2 \;\Rightarrow\; f \otimes f_1 \leq f \otimes f_2$

where $f_1 \leq f_2$ if and only if $f_1 \oplus f_2 = f_2$.

Clearly in a flow algebra all finite subsets $\{f_1, \ldots, f_n\}$ have a least upper bound, which is given by $\bar{0} \oplus f_1 \oplus \cdots \oplus f_n$.

Since the assumptions on a flow algebra are less demanding than in the case of idempotent semirings, we additionally introduce the notions of distributive and strict flow algebras.

Definition 3.2 A distributive flow algebra is a flow algebra $(F, \oplus, \otimes, \bar{0}, \bar{1})$, where $\otimes$ distributes over $\oplus$ on both sides, i.e.,

$$f_1 \otimes (f_2 \oplus f_3) = (f_1 \otimes f_2) \oplus (f_1 \otimes f_3)$$
$$(f_1 \oplus f_2) \otimes f_3 = (f_1 \otimes f_3) \oplus (f_2 \otimes f_3)$$

We also say that a flow algebra is strict if

$$\bar{0} \otimes f = \bar{0} = f \otimes \bar{0}$$

Fact 3.3 *Every idempotent semiring is a strict and distributive flow algebra.*

We consider flow algebras because they are closer to Monotone Frameworks and to other classical static analyses. Restricting our attention to semirings rather than flow algebras would mean restricting attention to strict and distributive frameworks. Note that the classical bit vector frameworks [59] are distributive, but not strict; hence they are not directly expressible using idempotent semirings.

Definition 3.4 A complete flow algebra is a flow algebra $(F, \oplus, \otimes, \bar{0}, \bar{1})$, where F is a complete lattice; we write $\bigoplus$ for the least upper bound. It is affine [59] if for all non-empty subsets $F' \neq \emptyset$ of F

$$f \otimes \bigoplus F' = \bigoplus \{f \otimes f' \mid f' \in F'\}$$
$$\bigoplus F' \otimes f = \bigoplus \{f' \otimes f \mid f' \in F'\}$$

Furthermore, it is completely distributive if it is affine and strict.

If the complete flow algebra satisfies the ascending chain condition [59] then it is affine if and only if it is distributive. The proof is analogous to the one presented in Appendix A of [59].

In general a good intuition behind flow algebras is to consider a complete lattice $L \to L$ of monotone functions over the complete lattice L. Then we can easily define a flow algebra for forward analyses by taking $(L \to L, \sqcup, \fatsemi, \lambda f.\bot, \lambda f.f)$ where $(f_1 \fatsemi f_2)(l) = f_2(f_1(l))$ for all $l \in L$. It is easy to see that all the laws of a complete flow algebra are satisfied. If we restrict the functions in $L \to L$ to be distributive, we obtain a distributive and complete flow algebra. Note that it can be used to define data-flow analyses such as reaching definitions [1, 59]. We look a bit more closely at some of such analyses below.

3.2.2 Example

As an example let us consider the family of forward "may" analyses that are instances of bit vector framework. They are generally defined in the following way:

- The lattice L is equal to $\mathcal{P}(D)$ for some finite D.
- The least upper bound operator is $\bigcup$.
- The transfer functions are monotone functions of the shape

  $$f_i(l) = (l \setminus k_i) \cup g_i$$

  where $k_i, g_i \in \mathcal{P}(D)$ correspond to the elements of D that are "killed" and "generated" at some program point i. This is also the source of a popular name for similar analyses — "kill/gen" analyses.
- The least element $\bot = \emptyset$.

In order to use such analyses with weighted pushdown systems we will construct a flow algebra $(\mathcal{F}, \oplus, \otimes, \bar{0}, \bar{1})$ that works on the level of functions $\mathcal{P}(D) \to \mathcal{P}(D)$. Since we are dealing here with a "kill/gen" analysis, this is actually quite easy — we express a function $f_i(l) = (l \setminus k_i) \cup g_i$ by a pair (k_i, g_i). Therefore, we have:

- $\mathcal{F} = \mathcal{P}(D) \times \mathcal{P}(D)$
- The $\oplus$ operator is defined as

$$f_1 \oplus f_2 = (k_1, g_1) \oplus (k_2, g_2) = (k_1 \cap k_2, g_1 \cup g_2)$$

- The $\otimes$ operator is defined as

$$f_1 \otimes f_2 = (k_1, g_1) \otimes (k_2, g_2) = (k_1 \cup k_2, (g_1 \setminus k_2) \cup g_2)$$

- $\bar{0} = (D, \emptyset)$
- $\bar{1} = (\emptyset, \emptyset)$

It should be easy to see that $\oplus$ is idempotent and commutative. Therefore, the semiring is naturally ordered with $f_1 \sqsubseteq f_2 \iff f_1 \oplus f_2 = f_2$. Furthermore, $\bar{0}$ is a neutral element for $\oplus$ and $\bar{1}$ is neutral for $\otimes$.

However, the interesting part is that $\bar{0}$ is not an annihilator for $\otimes$. Consider the following:

$$\begin{aligned}(D, \emptyset) \otimes (k, g) &= (D \cup k, (\emptyset \setminus k) \cup g) \\ &= (D, g)\end{aligned}$$

which clearly is not equal to $\bar{0}$ (unless $g = \emptyset$). Interestingly the annihilation works from the right:

$$\begin{aligned}(k, g) \otimes (D, \emptyset) &= (k \cup D, (g \setminus D) \cup \emptyset) \\ &= (D, \emptyset) \\ &= \bar{0}\end{aligned}$$

This makes perfect sense if we consider for a moment the classical transfer functions of such analyses. If we extend the ordering of $\mathcal{P}(D)$ pointwise to the monotone functions $\mathcal{P}(D) \to \mathcal{P}(D)$, the least element will be a function that always returns $\emptyset$, i.e., $f_\perp = \lambda l.\emptyset$. Clearly we have that:

$$\forall f : f_\perp \circ f = f_\perp$$

but in general it is not the case that:

$$\forall f : f \circ f_\perp = f_\perp$$

Therefore, such analyses do not naturally form a semiring, yet they fit quite well in the approach based on flow algebras.

3.3 Galois connections for flow algebras

Let us recall that *Galois connection* is a tuple (L, α, γ, M) such that L and M are complete lattices and α, γ are monotone functions (called abstraction and concretization functions) that satisfy $\alpha \circ \gamma \sqsubseteq \lambda m.m$ and $\gamma \circ \alpha \sqsupseteq \lambda l.l$. A *Galois insertion* is a Galois connection such that $\alpha \circ \gamma = \lambda m.m$. In this section we will present them in the setting of flow algebras.

In order to extend the Galois connections for flow algebras, we need to define what it means for a flow algebra to be an upper-approximation of another flow algebra. In other words we need to impose certain conditions on $\otimes$ operator and $\bar{1}$ element of the less precise flow algebra. The requirements are presented in the following definition.

Definition 3.5 For a Galois connection (L, α, γ, M) we say that the flow algebra $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ is an upper-approximation of $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ if

$$\alpha(\gamma(m_1) \otimes_L \gamma(m_2)) \sqsubseteq_M m_1 \otimes_M m_2$$
$$\alpha(\bar{1}_L) \sqsubseteq_M \bar{1}_M$$

If we have equalities in the above definition, then we say that the flow algebra $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ is *induced* from $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$.

Example 3.1 *Assume that we have a Galois connection (L, α, γ, M) between complete lattices L and M. We can easily construct $(L \to L, \alpha', \gamma', M \to M)$ which is a Galois connection between monotone function spaces over those lattices (for more details about this construction please see Section 4.4 of [59]), where α', γ' are defined as*

$$\alpha'(f) = \alpha \circ f \circ \gamma$$
$$\gamma'(g) = \gamma \circ g \circ \alpha$$

When both $(M \to M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ *and* $(L \to L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ *are forward analyses (as mentioned in Section 3.2.1), we have*

$$\begin{aligned}
\alpha'(\gamma'(g_1) \otimes_L \gamma'(g_2)) &= \alpha'((\gamma \circ g_1 \circ \alpha) \fatsemi (\gamma \circ g_2 \circ \alpha)) \\
&= \alpha'(\gamma \circ g_2 \circ \alpha \circ \gamma \circ g_1 \circ \alpha) \\
&\sqsubseteq \alpha'(\gamma \circ g_2 \circ g_1 \circ \alpha) \\
&= \alpha \circ \gamma \circ g_2 \circ g_1 \circ \alpha \circ \gamma \\
&\sqsubseteq g_2 \circ g_1 \\
&= g_1 \otimes_M g_2
\end{aligned}$$

$$\alpha'(\bar{1}_L) = \alpha \circ \lambda l.l \circ \gamma = \alpha \circ \gamma \sqsubseteq \lambda m.m = \bar{1}_M$$

Hence a flow algebra over $M \to M$ *is a upper-approximation of the flow algebra over* $L \to L$*. Note that in case of a Galois insertion the flow algebra over* $M \to M$ *is induced.*

Definition 3.5 requires a bit of care. Given a flow algebra $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ and a Galois connection (L, α, γ, M) it is tempting to define $\otimes_M$ by $m_1 \otimes_M m_2 = \alpha(\gamma(m_1) \otimes_L \gamma(m_2))$ and $\bar{1}_M$ by $\bar{1}_M = \alpha(\bar{1}_L)$. However, it is not generally the case that $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ will be a flow algebra. This motivates the following development.

Lemma 3.6 *Let* $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ *be a flow algebra,* (L, α, γ, M) *be a Galois insertion, define* $\otimes_M$ *by* $m_1 \otimes_M m_2 = \alpha(\gamma(m_1) \otimes_L \gamma(m_2))$ *and* $\bar{1}_M$ *by* $\bar{1}_M = \alpha(\bar{1}_L)$*. If*

$$\bar{1}_L \in \gamma(M) \quad \textit{and} \quad \otimes_L : \gamma(M) \times \gamma(M) \to \gamma(M)$$

then $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ *is a flow algebra (where* $\oplus_M$ *is* $\sqcup_M$ *and* $\bar{0}_M$ *is* $\bot_M$*).*

Proof. We need to ensure that $\otimes_M$ is associative:

$$\begin{aligned}
(m_1 \otimes_M m_2) \otimes_M m_3 &= \alpha(\gamma(\alpha(\gamma(m_1) \otimes_L \gamma(m_2))) \otimes_L \gamma(m_3)) \\
&= \alpha(\gamma(\alpha(\gamma(m'))) \otimes_L \gamma(m_3)) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(m_2) \otimes_L \gamma(m_3)) \\
m_1 \otimes_M (m_2 \otimes_M m_3) &= \alpha(\gamma(m_1) \otimes_L \gamma(\alpha(\gamma(m_2) \otimes_L \gamma(m_3)))) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(\alpha(\gamma(m')))) \\
&= \alpha(\gamma(m_1) \otimes_L \gamma(m_2) \otimes_L \gamma(m_3))
\end{aligned}$$

and similarly we need to show that $\bar{1}_M$ is a neutral element for $\otimes_M$

$$\begin{aligned}
\bar{1}_M \otimes m &= \alpha(\bar{1}_L) \otimes_M m \\
&= \alpha(\gamma(\alpha(\bar{1}_L)) \otimes_L \gamma(m)) \\
&= \alpha(\gamma(\alpha(\gamma(m'))) \otimes_L \gamma(m)) \\
&= \alpha(\gamma(m') \otimes_L \gamma(m)) \\
&= \alpha(\bar{1}_L \otimes_L \gamma(m)) \\
&= \alpha(\gamma(m)) \\
&= m
\end{aligned}$$

where $\bar{1}_L = \gamma(m')$ for some m'. The remaining properties of flow algebra hold trivially. □

The above requirements can be expressed in a slightly different way. This is presented by the following two lemmas.

Lemma 3.7 *For flow algebras* $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$, $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ *and a Galois insertion* (L, α, γ, M), *the following are equivalent:*

1. $\bar{1}_L = \gamma(\bar{1}_M)$
2. $\alpha(\bar{1}_L) = \bar{1}_M$ *and* $\bar{1}_L \in \gamma(M)$

Proof. The proof is available in Appendix A.1.

Lemma 3.8 *For flow algebras* $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$, $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ *and a Galois insertion* (L, α, γ, M), *the following are equivalent:*

1. $\forall m_1, m_2 : \gamma(m_1) \otimes_L \gamma(m_2) = \gamma(m_1 \otimes_M m_2)$
2. $\forall m_1, m_2 : \alpha(\gamma(m_1) \otimes_L \gamma(m_2)) = m_1 \otimes_M m_2$ *and* $\otimes_L : \gamma(M) \times \gamma(M) \to \gamma(M)$

Proof. The proof is available in Appendix A.2.

3.4 Program graphs

This section introduces program graphs, a representation of software (hardware) systems that is often used in model checking [3] to model concurrent and

distributed systems. Compared to the classical flow graphs [49, 59], the main difference is that in the program graphs the actions label the edges rather than the nodes.

Definition 3.9 A program graph over a space S has the form

$$(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$$

where

- Q is a finite set of states;
- Σ is a finite set of actions;
- $\rightarrow \subseteq \mathsf{Q} \times \Sigma \times \mathsf{Q}$ is a transition relation;
- $\mathsf{Q}_I \subseteq \mathsf{Q}$ is a set of initial states;
- $\mathsf{Q}_F \subseteq \mathsf{Q}$ is a set of final states; and
- $\mathcal{A} : \Sigma \rightarrow S$ specifies the meaning of the actions.

A concrete program graph is a program graph where $S = \mathbf{Dom} \hookrightarrow \mathbf{Dom}$, where **Dom** is the set of all configurations of a program, and $\mathcal{A} = \mathcal{T}$ where $\mathcal{T}$ is the semantic function. An abstract program graph is a program graph where S is a complete flow algebra.

Now we can define the collecting semantics [19, 59] of a concrete program graph in terms of a flow algebra. This can be used to establish the semantic correctness of an analysis by defining a Galois connection between the collecting semantics and the analysis.

Definition 3.10 We define the collecting semantics of a program graph using the flow algebra $(\mathcal{P}(\mathbf{Dom}) \rightarrow \mathcal{P}(\mathbf{Dom}), \cup, \fatsemi, \lambda.\emptyset, \lambda d.d)$, by

$$\mathcal{A}[\![a]\!](S) = \{\mathcal{T}[\![a]\!](s) \mid s \in S \wedge \mathcal{T}[\![a]\!](s) \text{ is defined}\}$$

where **Dom** is the set of all configurations of a program and $\mathcal{T}$ is the semantic function.

Now let us consider a number of processes each specified as a program graph $PG_i = (\mathsf{Q}_i, \Sigma_i, \rightarrow_i, \mathsf{Q}_{Ii}, \mathsf{Q}_{Fi}, \mathcal{A}_i, S)$ that are executed independently of one another except that they can exchange information via shared variables. The combined program graph $PG = PG_1 \mid\mid\mid \cdots \mid\mid\mid PG_n$ expresses the interleaving between n processes.

Definition 3.11 The interleaved program graph over S

$$\mathsf{PG} = \mathsf{PG}_1 \mid\mid\mid \cdots \mid\mid\mid \mathsf{PG}_n$$

is defined by $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, S)$ where

- $\mathsf{Q} = \mathsf{Q}_1 \times \cdots \times \mathsf{Q}_n$,
- $\Sigma = \Sigma_1 \uplus \cdots \uplus \Sigma_n$ (disjoint union),
- $\langle q_1, \cdots, q_i, \cdots, q_n \rangle \xrightarrow{a} \langle q_1, \cdots, q'_i, \cdots, q_n \rangle$ if $q_i \xrightarrow{a}_i q'_i$,
- $\mathsf{Q}_I = \mathsf{Q}_{I1} \times \cdots \times \mathsf{Q}_{In}$,
- $\mathsf{Q}_F = \mathsf{Q}_{F1} \times \cdots \times \mathsf{Q}_{Fn}$, and
- $\mathcal{A}[\![a]\!] = \mathcal{A}_i[\![a]\!]$ if $a \in \Sigma_i$.

Note that $\mathcal{A}_i : \Sigma_i \rightarrow S$ for all i and hence $\mathcal{A} : \Sigma \rightarrow S$.

Analogously to the previous definition, we say that a concrete interleaved program graph is an interleaved program graph where $S = \mathbf{Dom} \hookrightarrow \mathbf{Dom}$, and $\mathcal{A} = \mathcal{T}$ where $\mathcal{T}$ is the semantic function. An abstract interleaved program graph is an interleaved program graph where S is a complete flow algebra.

The application of this definition is presented in the example below, where we model the Bakery mutual exclusion algorithm. Note that the ability to create interleaved program graphs allows us to model concurrent systems using the same methods as in the case of sequential programs. This will be used to analyze and verify the algorithm in Section 3.7.

Example 3.2 *As an example we consider a variant of the Bakery algorithm for two processes. Let P_1 and P_2 be the two processes, and x_1 and x_2 be two shared variables both initialized to 0. The algorithm is as follows:*

```
do true ->
    x1 := x2 + 1;
    do ¬((x2 = 0) ∨ (x1 < x2)) ->
        skip
    od;
    critical section
    x1 := 0
od
```

```
do true ->
    x2 := x1 + 1;
    do ¬((x1 = 0) ∨ (x2 < x1)) ->
        skip
    od;
    critical section
    x2 := 0
od
```

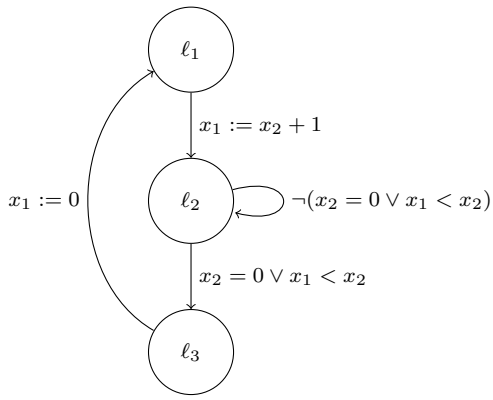


Figure 3.1: Program graph for the first process.

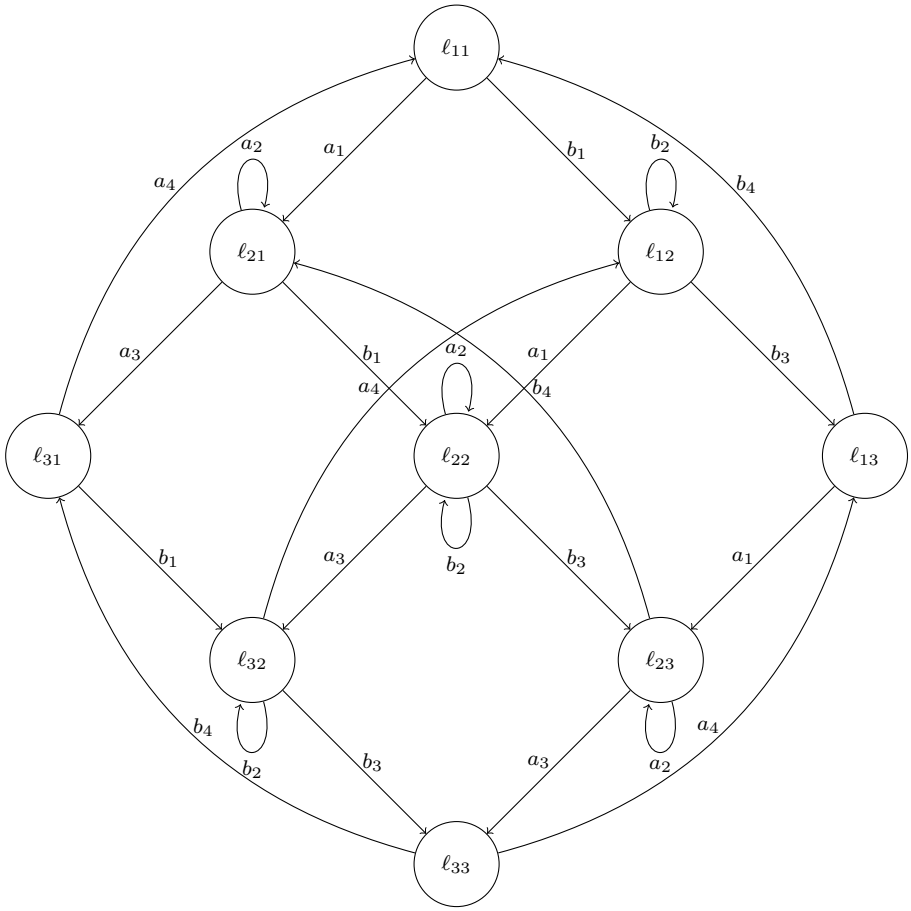
The variables x_1 and x_2 are used to resolve the conflict when both processes want to enter the critical section. When x_i is equal to zero, the process P_i is not in the critical section and does not attempt to enter it — the other one can safely proceed to the critical section. Otherwise, if both shared variables are non-zero, the process with smaller "ticket" (i.e., value of the corresponding variable) can enter the critical section. This reasoning is captured by the conditions of busy-waiting loops. When a process wants to enter the critical section, it simply takes the next "ticket" hence giving priority to the other process.

The program graph corresponding to the first process is quite simple and is presented in Figure 3.1 (the program graph for the second process is analogous). Now we can use the Definition 3.11 to obtain the interleaving of the two processes, which is depicted in Figure 3.2.

Since the result is also a program graph, it can be analysed in our framework.

3.5 Flow algebras over program graphs

Having defined flow algebras and program graphs, it remains to show how to obtain the analysis results. We shall consider two approaches, namely *MOP* and *MFP*. As already mentioned, these stand for Meet Over all Paths and Maximal Fixed Point, respectively. However, since we take a *join* (least upper bound) to merge information from different paths, in our setting these really mean join over all paths and least fixed point. However, we use the *MOP* and *MFP* acronyms for historical reasons.



$$
\begin{aligned}
a_1 :&\quad x_1 := x_2 + 1\\
a_2 :&\quad (x_2 = 0) \vee (x_1 < x_2)\\
a_3 :&\quad \neg((x_2 = 0) \vee (x_1 < x_2))\\
a_4 :&\quad x_1 := 0\\
b_1 :&\quad x_2 := x_1 + 1\\
b_2 :&\quad (x_1 = 0) \vee (x_2 < x_1)\\
b_3 :&\quad \neg((x_1 = 0) \vee (x_2 < x_1))\\
b_4 :&\quad x_2 := 0
\end{aligned}
$$

Figure 3.2: Interleaved program graph.

We consider the *MOP* solution first, since it is more precise and captures what we would ideally want to compute.

Definition 3.12 Given an abstract program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ over a complete flow algebra $(F, \oplus, \otimes, \bar{0}, \bar{1})$, and two sets $\mathsf{Q}_\circ \subseteq \mathsf{Q}$ and $\mathsf{Q}_\bullet \subseteq \mathsf{Q}$ we are interested in

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{\pi \in \mathsf{Path}(\mathsf{Q}_\circ, \mathsf{Q}_\bullet)} \mathcal{A}[\![\pi]\!]$$

where

$$\begin{aligned}\mathsf{Path}(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \{a_1 a_2 \cdots a_k \mid \ & \exists q_0, q_1, \cdots q_k : \\ & q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_k, \\ & q_0 \in \mathsf{Q}_\circ, q_k \in \mathsf{Q}_\bullet\}\end{aligned}$$

and

$$\mathcal{A}[\![a_1 a_2 \cdots a_k]\!] = \bar{1} \otimes \mathcal{A}[\![a_1]\!] \otimes \mathcal{A}[\![a_2]\!] \otimes \cdots \otimes \mathcal{A}[\![a_k]\!]$$

Since the *MOP* solution is not always computable (e.g., for Constant Propagation), one usually uses the *MFP* one which only requires that the lattice satisfies the Ascending Chain Condition and is defined as the least solution to a set of constraints. Let us first introduce those constraints.

Definition 3.13 Consider an abstract program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ over a complete flow algebra $(F, \oplus, \otimes, \bar{0}, \bar{1})$. This gives rise to a set $Analysis_F$ of constraints:

$$An_F^{\mathsf{Q}_\circ}(q) \sqsupseteq \begin{cases} \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus 1_F & \text{, if } q \in \mathsf{Q}_\circ \\ \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} & \text{, if } q \notin \mathsf{Q}_\circ \end{cases}$$

where $q \in \mathsf{Q}$, $\mathsf{Q}_\circ \subseteq \mathsf{Q}$.

We write $An_F^{\mathsf{Q}_\circ} \models Analysis_F$ whenever $An_F^{\mathsf{Q}_\circ} : \mathsf{Q} \rightarrow F$ is a solution to the constraints $Analysis_F$. Now we establish that there is always a least (i.e., best) solution of those constraints.

Lemma 3.14 *The set of solutions to the constraint system from Definition 3.13 is a Moore family (i.e., it is closed under $\bigsqcap$), which implies the existence of the least solution.*

Proof. The proof is in Appendix A.3.

Definition 3.15 We define *MFP* to be the least solution to the constraint system from Definition 3.13.

The following result states the general relationship between *MOP* and *MFP* solutions and shows in which cases the they coincide.

Proposition 3.16 *Consider the MOP and MFP solutions for an abstract program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ *over a complete flow algebra* $(F, \oplus, \otimes, \bar{0}, \bar{1})$*, then*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

If the flow algebra is affine and either $\forall q \in \mathsf{Q} : \mathsf{Path}(\mathsf{Q}_\circ, \{q\}) \neq \emptyset$ *or the flow algebra is strict then*

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

Proof. The proof is in Appendix A.4.

This is consistent with the previous results, e.g., for Monotone Frameworks, where the *MOP* and *MFP* coincide in case of distributive frameworks and otherwise *MFP* is a safe approximation of *MOP* [45].

3.6 Galois connections for program graphs

In the current section we show how the generalization of Galois connections to flow algebras can be used to upper-approximate solutions of the analyses. Namely, consider a flow algebra $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ and a Galois connection (L, α, γ, M). Moreover, let $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ be a flow algebra that is an upper-approximation of the flow algebra over L. We show that whenever we have a solution for an analysis in M then, when concretized, it is an upper-approximation of the solution of an analysis in L. First, we state necessary requirements for the analyses and then present the results for the *MOP* and *MFP* solutions.

3.6.1 Upper-approximation of Program Graphs

Since analyses using abstract program graphs are defined in terms of functions specifying effects of different actions, we need to impose conditions on these

functions.

Definition 3.17 Consider a flow algebra $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ that is an upper-approximation of $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ by a Galois connection (L, α, γ, M). A program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ is an upper-approximation of another program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ if

$$\forall a \in \Sigma : \alpha(\mathcal{A}[\![a]\!]) \sqsubseteq_M \mathcal{B}[\![a]\!]$$

It is quite easy to see that this upper-approximation for action implies one for paths.

Lemma 3.18 *If a program graph* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M)*, then for every path* π *we have that*

$$\mathcal{A}[\![\pi]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![\pi]\!])$$

Proof. The proof is in Appendix A.5.

Consider a flow algebra $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ induced from $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ by a Galois connection (L, α, γ, M). As in case of flow algebras, we say that a program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ is induced from $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ if we change the inequality from Lemma 3.18 to

$$\forall a \in \Sigma : \alpha(\mathcal{A}[\![a]\!]) = \mathcal{B}[\![a]\!]$$

3.6.2 Preservation of the *MOP* and *MFP* solutions

Now we will investigate what is the relationship between the solutions of an analysis in case of original program graph and its upper-approximation. Again we will first consider the *MOP* solution and then *MFP* one.

3.6.2.1 MOP

We want to show that if we calculate the *MOP* solution of the analysis $\mathcal{B}$ in M and concretize it (using γ), then we will get an upper-approximation of the *MOP* solution of $\mathcal{A}$ in L.

Lemma 3.19 *If a program graph defined by* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M) *then*

$$MOP_L(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \gamma(MOP_M(\mathsf{Q}_\circ, \mathsf{Q}_\bullet))$$

Proof. The result follows from Lemma 3.18. □

3.6.2.2 MFP

Let us now consider the *MFP* solution. We would like to prove that whenever we have a solution An_M of the constraint system $Analysis_M$ then, when concretized, it also is a solution to the constraint system $Analysis_L$. This is established by the following lemma.

Lemma 3.20 *If a program graph given by* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, M)$ *is an upper-approximation of* $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ *by* (L, α, γ, M) *then*

$$An_M^{\mathsf{Q}_\circ} \models Analysis_M \implies \gamma \circ An_M^{\mathsf{Q}_\circ} \models Analysis_L$$

and in particular

$$MFP_L^{\mathsf{Q}_\circ} \sqsubseteq \gamma \circ MFP_M^{\mathsf{Q}_\circ}$$

Proof. We consider only the case where $q \in \mathsf{Q}_\circ$ (the other case is analogous). From the assumption we have:

$$\begin{aligned}\gamma \circ An_M^{\mathsf{Q}_\circ} &\sqsupseteq \lambda q.\gamma\left(\bigoplus\{An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus \bar{1}_M\right)\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \gamma(\bar{1}_M)\end{aligned}$$

Now using the definition of upper-approximation of a flow algebra it follows that

$$\begin{aligned}&\lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q') \otimes \mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \gamma(\bar{1}_M)\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q')) \otimes \gamma(\mathcal{B}[\![a]\!]) \mid q' \xrightarrow{a} q\} \oplus \bar{1}_L\\ &\sqsupseteq \lambda q.\bigoplus\{\gamma(An_M^{\mathsf{Q}_\circ}(q')) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus \bar{1}_L\end{aligned}$$

We also know that every solution $An_L^{\mathsf{Q}_\circ}$ to the constraints $Analysis_L$ must satisfy

$$An_L^{\mathsf{Q}_\circ} \sqsupseteq \lambda q.\bigoplus\{An_L^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q\} \oplus \bar{1}_L$$

and it should be clear that $\gamma \circ An_M^{\mathsf{Q}_\circ}$ is also a solution these constraints. □

3.7 Application to the bakery algorithm

In this section we use flow algebras and Galois insertions to verify the correctness of the Bakery mutual exclusion algorithm. Although the Bakery algorithm is designed for an arbitrary number of processes, we consider the simpler setting with two processes, as in Example 3.2. For reader's convenience we recall the pseudo-code of the algorithm:

```
do true ->
    x1 := x2 + 1;
    do ¬((x2 = 0) ∨ (x1 < x2)) ->
        skip
    od;
    critical section
    x1 := 0
od
```

```
do true ->
    x2 := x1 + 1;
    do ¬((x1 = 0) ∨ (x2 < x1)) ->
        skip
    od;
    critical section
    x2 := 0
od
```

We want to verify that the algorithm ensures mutual exclusion, which is equivalent to checking whether the state $(3, 3)$ (corresponding to both processes being in the critical section at the same time) is unreachable in the interleaved program graph. First we define the collecting semantics, which tells us the potential values of the variables x_1 and x_2. Since they can never be negative, we take the complete lattice to be the monotone function space over $\mathcal{P}(\mathbb{Z} \times \mathbb{Z})$. This gives rise to the flow algebra $\mathcal{C}$ of the form

$$(\mathcal{P}(\mathbb{Z} \times \mathbb{Z}) \to \mathcal{P}(\mathbb{Z} \times \mathbb{Z}), \cup, \fatsemi, \lambda ZZ.\emptyset, \lambda ZZ.ZZ)$$

The semantic function is defined as follows

$$\begin{array}{lcl}
\mathcal{T}[\![x_1 := x_2 + 1]\!] & = & \lambda ZZ.\{(z_2 + 1, z_2) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_2 := x_1 + 1]\!] & = & \lambda ZZ.\{(z_1, z_1 + 1) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_1 := 0]\!] & = & \lambda ZZ.\{(0, z_2) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![x_2 := 0]\!] & = & \lambda ZZ.\{(z_1, 0) \mid (z_1, z_2) \in ZZ\} \\
\mathcal{T}[\![e]\!] & = & \lambda ZZ.\{(z_1, z_2) \mid \mathcal{E}[\![e]\!](z_1, z_2) \wedge (z_1, z_2) \in ZZ\}
\end{array}$$

where $\mathcal{E} : Expr \to (\mathbb{Z} \times \mathbb{Z} \to \{true, false\})$ is used for evaluating expressions. Unfortunately, as the values of x_1 and x_2 may grow unboundedly, the underlying transition system of the parallel composition of two processes is infinite. Hence it is not possible to naively use it to verify the algorithm.

Therefore, we introduce some abstraction. Using our approach we would like to define an analysis that is an upper-approximation of the collecting semantics. This should allow us to compute the result and at the same time guarantee that the analysis is semantically correct. The only remaining challenge is to define a domain that is precise enough to capture the property of interest and then show that the analysis is an upper-approximation of the collecting semantics.

For our purposes it is enough to record when the conditions allowing to enter the critical section (e.g. $(x_2 = 0) \vee (x_1 < x_2)$) are true or false. For that we can use the Sign Analysis. We take the complete lattice to be the monotone function space over $\mathcal{P}(\mathcal{S} \times \mathcal{S} \times \mathcal{S})$, where $\mathcal{S} = \{-, \bar{0}, +\}$. The three components record the signs of variables x_1, x_2 and their difference i.e. $x_1 - x_2$, respectively. We define a Galois connection $(\mathcal{P}(\mathbb{Z} \times \mathbb{Z}), \alpha, \gamma, \mathcal{P}(\mathcal{S} \times \mathcal{S} \times \mathcal{S}))$ by the extraction function

$$\eta(z_1, z_2) = (sign(z_1), sign(z_2), sign(z_1 - z_2))$$

where

$$sign(z) = \begin{cases} - & \text{if } z < 0 \\ \bar{0} & \text{if } z = 0 \\ + & \text{if } z > 0 \end{cases}$$

Then α and γ are defined by

$$\alpha(ZZ) = \{\eta(z_1, z_2) \mid (z_1, z_2) \in ZZ\}$$
$$\gamma(SSS) = \{(z_1, z_2) \mid \eta(z_1, z_2) \in SSS\}$$

for $ZZ \subseteq \mathbb{Z} \times \mathbb{Z}$ and $SSS \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$.

However, note that the set $\mathcal{P}(\mathcal{S} \times \mathcal{S} \times \mathcal{S})$ contains superfluous elements, such as $(0, 0, +)$. Therefore, we reduce the domain of the Sign Analysis to the subset that contains only meaningful elements. For that purpose we use the already defined extraction function η. The resulting set $\mathcal{P}(\eta[\mathbb{Z} \times \mathbb{Z}])$ is defined using

$$\eta[\mathbb{Z} \times \mathbb{Z}] = \{\eta(z_1, z_2) \mid (z_1, z_2) \in \mathbb{Z} \times \mathbb{Z}\}$$

It is easy to see that

$$\eta[\mathbb{Z} \times \mathbb{Z}] = \left\{ \begin{array}{c} (\bar{0}, \bar{0}, \bar{0}), (\bar{0}, +, -), (+, \bar{0}, +), \\ (+, +, \bar{0}), (+, +, +), (+, +, -) \end{array} \right\}$$

This gives rise to a Galois insertion (recall Example 3.1)

$$(\mathcal{P}(\mathbb{Z} \times \mathbb{Z}) \to \mathcal{P}(\mathbb{Z} \times \mathbb{Z}), \alpha', \gamma', \mathcal{P}(\eta[\mathbb{Z} \times \mathbb{Z}]) \to \mathcal{P}(\eta[\mathbb{Z} \times \mathbb{Z}]))$$

where:

$$\alpha'(f) = \alpha \circ f \circ \gamma$$

$$\gamma'(g) = \gamma \circ g \circ \alpha$$

We next consider the flow algebra $\mathcal{S}$ given by

$$(\mathcal{P}(\eta[\mathbb{Z} \times \mathbb{Z}]) \to \mathcal{P}(\eta[\mathbb{Z} \times \mathbb{Z}]), \cup, ⨾, \lambda SSS.\emptyset, \lambda SSS.SSS)$$

and note that it is induced from the flow algebra $\mathcal{C}$ by the Galois insertion (for details refer to Example 3.1). Now we can induce transfer functions for the Sign Analysis. As an example let us consider the case of $x_2 := 0$ and calculate

$$\begin{aligned}
\mathcal{A}[\![x_2 := 0]\!](SSS) &= \alpha(\mathcal{T}[\![x_2 := 0]\!](\gamma(SSS))) \\
&= \alpha(\mathcal{T}[\![x_2 := 0]\!](\{(z_1, z_2) \mid \eta(z_1, z_2) \in SSS\})) \\
&= \alpha(\{(z_1, 0) \mid \eta(z_1, z_2) \in SSS\}) \\
&= \{(s_1, 0, s_1) \mid (s_1, s_2, s) \in SSS\}
\end{aligned}$$

Other transfer functions are induced in a similar manner and are omitted.

Clearly the program graph over flow algebra $\mathcal{S}$ is an upper-approximation of the concrete program graph (i.e., one defined over the collecting semantics). It follows that the Sign Analysis is semantically correct. Therefore, we can safely use it to verify the correctness of the Bakery algorithm. For the actual calculations of the least solution for the analysis problem we use the *Succinct Solver* [60], in particular its latest version [29] that is based on Binary Decision Diagrams [12]. The analysis can be expressed in $ALFP$ (Alternation-free Least Fixed Point logic, i.e., the constraint language of the solver) and, as expected, the result obtained for the node $(3, 3)$ is the empty set, which means that the node is unreachable. Thus the mutual exclusion property is guaranteed.

3.8 Conclusions

In this chapter we presented a general framework that uses program graphs and flow algebras to define analyses. One of our main contributions is the introduction of flow algebras, which are algebraic structures less restrictive than idempotent semirings. Their main advantage and our motivation for introducing them is the ability to directly express the classical analyses, which is clearly not possible when using idempotent semirings. Moreover, the presented approach has certain advantages over Monotone Frameworks, such as the ability to handle both sequential and concurrent systems in the same manner. We also define both *MOP* and *MFP* solutions and establish that the classical result of their coincidence in case of distributive analyses carries over to our framework.

Furthermore, we investigated how to use Galois connections in this setting. They are a well-known and powerful technique that is often used to "move" from a

computationally costly analysis to a less expensive one. Our contribution is the use of Galois connections to upper-approximate and induce flow algebras and program graphs. This allows creating new analyses such that their semantic correctness is preserved.

We also demonstrated our approach by applying it to a variant of the Bakery mutual exclusion algorithm. We verified its correctness by moving from a precise, but uncomputable analysis to the one that is both precise enough for our purposes and easily computable. Since the analysis is induced from the collecting semantics by a Galois insertion, we can be sure that it is semantically correct.

Finally, it should be noted that in this chapter we only considered an intraprocedural analysis. In Chapter 4 we review some techniques for interprocedural analysis focusing mainly on communicating and weighted pushdown systems and briefly discuss how those techniques link back to this chapter and the concept of flow algebras. Then we show in Chapter 5 that flow algebras can be used directly with pushdown systems.

Chapter 4

Interprocedural analysis

In this chapter we will briefly present some preliminaries and previous work in the area of interprocedural analysis. We will start with some introduction to interprocedural analysis in Section 4.1. This will set the scene for the introduction and discussion of *pushdown systems* along with some of their extensions. In Section 4.2 we will present the basic definitions and then go on to present the *weighted pushdown systems* in Section 4.3 as well as *communicating pushdown systems* in Section 4.4. We will also have a brief discussion on the similarities and some minor differences between the weighted and communicating pushdown systems. Finally, we will show that some of the concepts we developed in Chapter 3 are useful in the context of weighted and communicating pushdown systems.

4.1 Introduction

Static analysis algorithms usually operate on a program representation called *control flow graph* (CFG), which consists of a set of basic blocks and edges connecting them. Basic blocks usually correspond to some statement (or sequence of statements) and the edges represent the control flow. CFGs are usually used to represent a single procedure and it is often assumed that it has unique *entry*

and *exit* blocks. To represent a program consisting of multiple procedures an *interprocedural control flow graph* (ICFG) can be used. It consists of a number of CFGs connected by *call-edges* which go from the call site to the entry point of the procedure and *return-edges* that connect the exit and the return points.

The presence of procedures is an interesting challenge for static analysis, especially when recursion is allowed. Without recursion it is possible to simply inline all procedures and thus transform the program to one without procedures and then perform intraprocedural analysis. The price for this approach is that it can result in an exponential blowup of the size of the program, which can make the analysis problem intractable. What is more, if we allow recursion, this simple approach is not possible, because the inlining procedure itself can fail to terminate. Furthermore, an analysis cannot explicitly track the contents of the stack since it can grow arbitrarily.

A naïve approach to the problem can simply ignore the stack or any calling context and treat both the call- and return-edges as usual CFG edges. However, analyses based on this approach, known as context-insensitive analyses, are quite imprecise. Intuitively this is due to the fact that by ignoring the context of procedure calls, they are not able to distinguish between different calls. A very clear example of this problem is that they are not capable of matching procedure calls with their returns. In other words, such analyses will also consider paths that are not possible at runtime, thus making the result of the analysis far less precise.

A possible remedy for this problem is to take the calling context into account. Analyses that do so are called context-sensitive and achieve better precision, but often at the cost of efficiency. One of the most influential works concerning the interprocedural analysis is due to Sharir and Pnueli [73], where two approaches to interprocedural analysis were proposed: functional approach and so-called call strings. We will briefly describe the former since it is quite relevant in the context of weighted pushdown systems. The idea behind the functional approach to interprocedural analysis is that every statement can be considered as a state transformer that describes how it changes the state of the program (i.e., if we have a lattice L consider the functions $L \to L$). A sequence of statements can then be considered as a composition of their corresponding transformers and merging of various branches as the least upper bound of the summarizations corresponding to those branches. With this, one can transform an interprocedural program into a set of non-linear equations, which when solved provide the summarization of each procedure (i.e., the changes from the entry to the exit point of the procedure). Having that information it is not difficult to compute the state transformer summarizing the changes from the beginning of the program to any of its points. Then one can apply some initial information to this summarization to obtain the final result (in this way one can reuse the summa-

rization for different initial contexts). Moreover, it should be emphasized that once the first set of equations is solved, it can be reused, i.e., the procedures themselves are analyzed only once. However, there are also some downsides to this approach. One of the most serious ones is that some analyses represented in terms of functions might fail to satisfy certain useful properties, e.g., the ascending chain condition. Furthermore, efficient representation of those transformers can be quite challenging itself.

Some other well-known results in the area of interprocedural analysis include the interprocedural coincidence theorem in the presence of local variables due to Knoop and Steffen [50]. Another significant contribution is by Cousot and Cousot [18] that give the abstract interpretation [19] based approach to interprocedural analysis. As well as paper by Sagiv, Reps and Horwitz [70] where the taken approach reduces the interprocedural analysis to a graph reachability problem.

Finally, it is interesting to discuss where the pushdown systems enter the picture. Commonly many interprocedural analyses will merge together information from all calling contexts for any given point in a procedure. In other words once we have the results of the analysis it is not possible to recover the dataflow information for some specific calling context (e.g., a specific stack). Pushdown systems allow exactly that, in fact they even make it possible to pose queries with respect to regular languages of stack contents. Therefore, the WPDS can be considered as a generalization of the functional approach to interprocedural analysis.

4.2 Pushdown systems

Pushdown systems [9, 26, 72] and weighted pushdown systems [66, 67] have recently become one of the most common techniques for describing and representing recursive programs. They have been used for verification purposes in many different projects and contexts. The examples include the Moped [72] and jMoped [75] model checkers that extensively use pushdown systems or Codesurfer [4] that takes advantage of weighted pushdown systems. What is more, weighted pushdown systems have been used also in scenarios different than recursive programs, for instance, to model SPKI/SDSI authorization framework [42, 43] or for process calculi with aspect-oriented features [76] (which will be also discussed in Chapter 6). In this section we will recall some of the basic definitions of pushdown systems as well present the main results and try to provide some intuition behind them. The section mostly follows [9, 26, 72].

Definition 4.1 A pushdown system is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of control locations, Γ is a finite set of stack symbols and Δ is a finite set of pushdown rules of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, where $w \in \Gamma^*$ and $|w| \leq 2$.

Note that the requirement $|w| \leq 2$ is not a serious restriction and any pushdown system can be transformed to satisfy it. This can be achieved by adding some fresh control locations and pushing $|w|$ in a few steps. Nevertheless, this is often unnecessary since the three rules are enough to encode control flow of a recursive program:

- $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \rangle$ corresponds to ordinary statement that goes from location γ_1 to γ_2,
- $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$ corresponds to a procedure call where the entry point of the procedure is γ_2 and the procedure should return to the location γ_3,
- $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \epsilon \rangle$ corresponds to returning from a procedure.

To get some intuition behind the pushdown systems, note that they can be used in a very natural way to describe *boolean programs*, that is procedural programs (possibly with recursion) where all variables are of type Boolean. Boolean programs have been successfully used in Bebop tool [6] which is used in the SLAM toolkit [7, 5] for verifying device drivers.

Example 4.1 *To represent Boolean programs in terms of a pushdown system, one can encode the global variables in the control locations and local variables in the stack contents. For instance an assignment when going from node n_1 to node n_2 can be expressed using the following pushdown rule:*

$$\langle g_1, (n_1, l_1) \rangle \hookrightarrow \langle g_2, (n_2, l_2) \rangle$$

where g_1 and g_2 (l_1 and l_2 represent the state of global (local) variables before and after the assignment. The pop rule is quite easy — we might need to update the global variables, but all the local information can be popped from the stack.

$$\langle g_1, (n_1, l_1) \rangle \hookrightarrow \langle g_2, \epsilon \rangle$$

Finally, the push rule can be defined as follows:

$$\langle g_1, (n_1, l_1) \rangle \hookrightarrow \langle g_2, (n_2, l_2)(n_3, l_3) \rangle$$

where l_2 are the local variables of the called procedure and l_3 are the variables saved on the stack (and thus will be restored by popping the stack).

One of the most important concepts when discussing the pushdown systems is that of a *configuration*.

Definition 4.2 A configuration of a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ is a pair $\langle p, s \rangle$ where $p \in P$ and $s \in \Gamma^*$.

As an example, a configuration can be thought of as the state or description of an abstract program at some point during its execution — the control location describing the global state and the stack which provides the current location along with all the return addresses (corresponding to the procedure that have been called).

Clearly a pushdown system gives rise to a (possibly infinite) transition systems, where we can move between configurations using the pushdown rules. The transition relation for this system is defined more formally below. For every pushdown rule $r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, w \rangle$ we have

$$\langle p_1, \gamma s \rangle \overset{r}{\Longrightarrow} \langle p_2, ws \rangle$$

for all $s \in \Gamma^*$. Sometimes we will omit the annotation of the specific pushdown rule — this means that we assume there exists a rule that allows moving between the given configurations. The reflexive, transitive closure of $\Longrightarrow$ will be denoted as $\Longrightarrow^*$ (and annotated with sequences of pushdown rules). Having a precise definition of the transition relation (and its reflexive transitive closure) allows us to define the concepts of successor and predecessor configurations. We call a configuration c_2 an immediate successor (predecessor) of c_1 if $c_1 \Longrightarrow c_2$ ($c_2 \Longrightarrow c_1$). Similar to immediate successors (predecessors) one can also define the general successors (predecessors) using the $\Longrightarrow^*$, namely a configuration c_2 a successor (predecessor) of c_1 if $c_1 \Longrightarrow^* c_2$ ($c_2 \Longrightarrow^* c_1$).

In many verification problems it is desirable to talk about the sets of successors or predecessors of a given configuration or set of configurations. They are often denoted as $Post^*(C)$ and $Pre^*(C)$ respectively, where C is some set of configuration. More formally:

$$Post^*(C) = \{c_2 \mid c_1 \Longrightarrow^* c_2, c_1 \in C\}$$
$$Pre^*(C) = \{c_2 \mid c_2 \Longrightarrow^* c_1, c_1 \in C\}$$

Note that those sets can be in general infinite (even if C is finite). In order to compute the sets of successors and predecessor we need some symbolic representation. This leads to the following definition:

Definition 4.3 Given a pushdown system $\mathcal{P} = (P, \Gamma, \Delta)$ a $\mathcal{P}$-automaton is a tuple $(Q, \Gamma, \rightarrow, P, F)$, where:

- Q is a finite set of states such that $P \subseteq Q$
- $\rightarrow \subseteq Q \times \Gamma \times Q$ is a finite set of transitions
- $P \subseteq Q$ is a finite set of initial states
- $F \subseteq Q$ is a finite set of final states

We define a transition relation $\rightarrow^* : Q \times \Gamma^* \times Q$ that is the smallest relation such that:

- $q \xrightarrow{\epsilon}{}^* q$ for every $q \in Q$
- if $q_1 \xrightarrow{\gamma} q_2$ then $q_1 \xrightarrow{\gamma}{}^* q_2$ if
- if $q_1 \xrightarrow{\gamma_1} q_2$ and $q_2 \xrightarrow{\gamma_2} q_3$ then $q_1 \xrightarrow{\gamma_1\gamma_2}{}^* q_3$.

We say that a $\mathcal{P}$-automaton accepts a configuration $\langle p, s\rangle$ if and only if $p \xrightarrow{s}{}^* q$ where $q \in F$. Moreover, a set of configurations is regular if it is accepted by some $\mathcal{P}$-automaton. Finally, one of the crucial results in the pushdown systems says that the sets of successors or predecessors of a regular set of configurations are regular themselves [9, 26, 72]. This is essential since it guarantees that we can always represent those sets as $\mathcal{P}$-automata.

Therefore, the algorithms for Pre^* and $Post^*$ take as input a pushdown system and an initial automaton $\mathcal{A}$ that represents the set of configurations whose predecessors or successors we want to compute. Both algorithms are basically saturation procedures, i.e., they keep adding new transitions to $\mathcal{A}$ according to some rule, until no further transitions can be added. Since the number of possible transitions is finite (in Pre^* the algorithm does not add any new states, and in $Post^*$ always a bounded number of them), the algorithms must terminate and return the $\mathcal{A}_{pre^*}$ or $\mathcal{A}_{post^*}$, which represent the possibly infinite number of reachable configurations. Additionally, the procedures assume that the initial automaton $\mathcal{A}$ does not contain any transitions into initial states, nor any ϵ-transitions.

We will start with the computation of $\mathcal{A}_{pre^*}$ as it is slightly easier. The procedure can be expressed in just one rule:

> If $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, w\rangle \in \Delta$ and $p_2 \xrightarrow{w}{}^* q$ in the current automaton (for some $q \in Q$) then add a transition $p_1 \xrightarrow{\gamma} q$.

Note that the procedure does not add any new states, nor any ϵ-transitions.

It might be helpful to consider the following example, which should provide some intuition behind the rule.

Example 4.2 *Let us have a pushdown rule $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, w\rangle$ and a configuration $\langle p_2, ws\rangle$ (where $s \in \Gamma^*$) that is a predecessor of some configuration of interest c, namely there exists σ such that $\langle p_2, ws\rangle \overset{\sigma}{\Longrightarrow}{}^* c$ and thus in the automaton we have:*

$$p_2 \xrightarrow{w}{}^* q \xrightarrow{s}{}^* q_f$$

where $q \in Q$ and $q_f \in F$. But the configuration $\langle p_1, \gamma s\rangle$ is clearly also a predecessor of c, since $\langle p_1, \gamma s\rangle \overset{r}{\Longrightarrow} \langle p_2, ws\rangle \overset{\sigma}{\Longrightarrow}{}^* c$. So we should add a transition $p_1 \xrightarrow{\gamma} q$ to have:

$$p_1 \xrightarrow{\gamma} q \xrightarrow{s}{}^* q_f$$

For the actual algorithm we refer the reader to [26], here we recall only its time complexity: $\mathcal{O}(|\Delta||Q|^2)$ and space complexity: $\mathcal{O}(|\Delta||Q| + |\rightarrow|)$.

The procedure for computing $Post^*$ is a bit more complex. Due to how the $Post^*$ algorithm works, we will use the reverse arrow notation for the transitions of the automata, i.e., we will write $q \xleftarrow{\gamma} p$ for the transition that we previously denoted by $p \xrightarrow{\gamma} q$. This will be useful when discussing the relationship between the transition system that arises due to the pushdown rules and the $\mathcal{A}_{post^*}$ automata (see also the below example). Apart from that, one of the main differences compared to Pre^* is the fact that it does add some new states as well as ϵ-transitions to the automaton. Thus, for clarity we additionally denote ${}^*\xleftarrow{\epsilon} \circ \xleftarrow{\gamma} \circ {}^*\xleftarrow{\epsilon}$ as $\overset{\gamma}{\dashleftarrow}$. Finally, the whole procedure is performed in in two stages:

1. For every pushdown rule $r \in \Delta$ such that r is of the form

   $$\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3\rangle$$

   add to the automaton a new state r and a transition $r \xleftarrow{\gamma_2} p_2$.[1]

2. Perform the saturation procedure according to the following rules:

   - If $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, \epsilon\rangle$ and $q \overset{\gamma}{\dashleftarrow} p_1$ in the current automaton add a new transition $q \xleftarrow{\epsilon} p_2$.

[1] It is interesting to note that some formulations of the procedure add a state q_{p_2,γ_2}. We will discuss this difference in Section 4.5.

- If $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2 \rangle$ and $q \overset{\gamma_1}{\dashleftarrow} p_1$ in the current automaton add a new transition $q \overset{\gamma_2}{\longleftarrow} p_2$.
- If $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$ and $q \overset{\gamma_1}{\dashleftarrow} p_1$ in the current automaton add a new transition $q \overset{\gamma_3}{\longleftarrow} r$.

The additional complexity if $Post^*$ is due to the fact that we have three cases of the right hand of a pushdown rule (i.e., $w = \epsilon$, $w = \gamma$ or $w = \gamma_1\gamma_2$). Since the most interesting case is for $w = 2$ let us consider how the procedure handles this situation. Hopefully this example will also show that using the "reversed" arrows for automata transitions is quite intuitive in this scenario.

Example 4.3 *Consider a pushdown rule $r_1 = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$ and a configuration $\langle p_1, \gamma_1 s \rangle$ (where $s \in \Gamma$) that it is a successor of some configuration of interest c, namely there exists σ such that $c \overset{\sigma}{\Longrightarrow}{}^* \langle p_1, \gamma_1 s \rangle$ and thus we have in the automaton*

$$q_f \overset{*}{\longleftarrow}{}^{s} q \overset{\gamma_1}{\longleftarrow} p_1$$

where $q \in Q$ and $q_f \in F$. But the configuration $\langle p_2, \gamma_2\gamma_3 s \rangle$ is also a successor of c, since

$$c \overset{\sigma}{\Longrightarrow}{}^* \langle p_1, \gamma_1 s \rangle \overset{r_1}{\Longrightarrow} \langle p_2, \gamma_2\gamma_3 s \rangle$$

In order to model the fact that the stack grows, we add the additional state and have:

$$q_f \overset{*}{\longleftarrow}{}^{s} q \overset{\gamma_3}{\longleftarrow} r_1 \overset{\gamma_2}{\longleftarrow} p_2$$

Now consider the case when there is also a rule saying $r_2 = \langle p_2, \gamma_2 \rangle \hookrightarrow \langle p_3, \epsilon \rangle$. In this situation $\langle p_3, \gamma_3 s \rangle$ we will also be a successor since we have:

$$c \overset{\sigma}{\Longrightarrow}{}^* \langle p_1, \gamma_1 s \rangle \overset{r_1}{\Longrightarrow} \langle p_2, \gamma_2\gamma_3 s \rangle \overset{r_2}{\Longrightarrow} \langle p_3, \gamma_3 s \rangle$$

This is modeled using the ϵ-transition, i.e., we have

$$q_f \overset{*}{\longleftarrow}{}^{s} q \overset{\gamma_3}{\longleftarrow} r_1 \overset{\epsilon}{\longleftarrow} p_3$$

Finally, as noted in [10, 11] if $\mathcal{A}$ satisfies our requirements, then $\overset{\gamma}{\dashleftarrow}$ is actually equal to $\overset{\gamma}{\longleftarrow} \circ \overset{\epsilon}{\longleftarrow} \cup \overset{\gamma}{\longleftarrow}$. To see why this holds consider the following facts:

- there are no ϵ transitions in the original automaton,
- the original automaton does not have any transitions going to initial states and the saturation procedure does not add such transitions (i.e., a transition from an initial state always goes to some non-initial state),

- the saturation procedure only adds ϵ-transitions going from initial states.

Therefore, an ϵ-transition is only possible from some initial state, but since only the first state in a run of the automaton is initial, then we cannot make more than one ϵ-transition.

Again for the actual algorithm we refer the reader to [26], here we recall only its time and space complexity: $\mathcal{O}(|P||\Delta|(|\Delta| + |Q|) + |P|| \to |)$.

4.3 Weighted pushdown systems

One of the limitations of pushdown systems is that they require finite abstractions — both the set of control locations and stack alphabet must be finite. This makes it difficult to use them in situations that required infinite abstractions. This problem was solved in [66, 67, 65, 53] by introducing *weighted pushdown systems*, which are additionally equipped with an abstract domain that forms a bounded, idempotent semiring. Every pushdown rule is associated with some value from this domain and they are used during the Pre^* and $Post^*$ procedures to produce a weighted automaton. The main restriction on the semiring is that it must not contain infinite ascending chains, which makes it possible to calculate the least solution using fixpoint approach based on Kleene iteration.

A similar development in [10, 11] uses almost the same idea but additionally allows idempotent semirings that might contain infinite ascending chains, but where $\otimes$ operator is commutative. In this section we will focus on weighted pushdown systems and discuss the second approach in Section 4.4.

Definition 4.4 A weighted pushdown system (WPDS) is a tuple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P}$ is a pushdown system, $\mathcal{S} = (S, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring and $f : \Delta \to S$ maps pushdown rules to the elements of S.

Now we can use the fact that every pushdown rule has a semiring weight to define the weight of a sequence of pushdown rules. Let $\sigma = [r_1, \ldots, r_n] \in \Delta^*$ be such a sequence, then we define $v(\sigma) = f(r_1) \otimes \cdots \otimes f(r_n)$.

It is important to recall that because of the unbounded stack, a PDS gives rise to an infinite graph of configurations. At the same time WPDS aim to calculate the semiring value for each of the potentially infinite configurations. Ordinary pushdown systems do not include any weights and solve the problem

by creating an automaton that represents all successor or predecessor configurations. Similarly, WPDS achieve their goal by using a weighted automaton, i.e., where every transition of the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ is additionally annotated with a semiring value. This makes it possible to compute the result for any given configuration (or even regular set of configurations) by reading the weights from the automaton. We will try to give some more intuition behind this approach by reviewing the WPDS saturation rules for both *Pre** and *Post** procedures.

More formally, WPDS allow us to compute:

- in case of predecessors of some regular set of configurations C

  $$\delta(c_1) = \bigoplus\{v(\sigma) \mid c_1 \overset{\sigma}{\Longrightarrow}{}^* c_2, c_2 \in C\}$$

  i.e., what is the semiring value of all the paths going from configuration c_1 to some configuration in C,

- in case of successors of some regular set of configurations C

  $$\delta(c_1) = \bigoplus\{v(\sigma) \mid c_2 \overset{\sigma}{\Longrightarrow}{}^* c_1, c_2 \in C\}$$

  i.e., what is the semiring value of all the paths going from some configuration in C to c_1.

The biggest advantage of this extension is that the only requirement is that the domain forms an idempotent semiring with no infinite ascending chains. Thus, infinite abstractions are possible.

First, let us look at the rule for *Pre**.

> If $r = \langle p_1, \gamma\rangle \hookrightarrow \langle p_2, w\rangle$ and $p_2 \overset{w}{\longrightarrow}{}^* q$ with weight d in the current automaton (for some $q \in Q$) then add a transition $p_1 \overset{\gamma}{\rightarrow} q$ with weight $f(r) \otimes d$.

The rule is almost the same as in case of PDS — the only difference is the addition of semiring weights.

Example 4.4 *Let us have a pushdown rule $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, w\rangle$ and a configuration $\langle p_2, ws\rangle$ (where $s \in \Gamma^*$) that it is a predecessor of some configuration of interest c, namely there exists σ such that $\langle p_2, ws\rangle \overset{\sigma}{\Longrightarrow}{}^* c$ with weight d_1 and thus in the automaton we have:*

$$p_2 \overset{w}{\longrightarrow}{}^* q \overset{s}{\rightarrow}{}^* q_f$$

where $q \in Q$ and $q_f \in F$. As before it is clear that configuration $\langle p_1, \gamma s\rangle$ is also a predecessor of c, since $\langle p_1, \gamma s\rangle \overset{r}{\Longrightarrow} \langle p_2, ws\rangle \overset{\sigma}{\Longrightarrow}{}^* c$. So we add a transition to have:

$$p_1 \overset{\gamma}{\rightarrow} q \overset{s}{\rightarrow}{}^* q_f$$

but the weight of $p_1 \overset{\gamma}{\rightarrow} q$ should be $f(r) \otimes d_1$ to express that we additionally perform r_1 at the beginning.

Now if the path $q \overset{s}{\rightarrow}{}^* q_f$ has weight d_2 then the weight of getting from $\langle p_1, \gamma s\rangle$ to some configuration of interest would be $f(r) \otimes d_1 \otimes d_2$.

It is important to note here the order in which we "multiply" the weights of the transitions — we do that in the order of transitions that are taken. We emphasize this because it is an important difference when compared to the way one treats the $\mathcal{A}_{post^*}$ automaton.

As in the case of plain PDS the procedure for *Post** is slightly more complex. Nevertheless, the basic mechanism is almost the same and the only thing that requires additional attention is how we compute the weights.

1. For every pushdown rule $r \in \Delta$ such that r is of the form $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3\rangle$ add to the automaton a new state q_{p_2,γ_2}
2. Perform the saturation procedure according to the following rules:
   - If $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, \epsilon\rangle$ and $q \overset{\gamma}{\dashleftarrow} p_1$ is in the current automaton with weight d add a new transition $q \overset{\epsilon}{\leftarrow} p_2$ with weight $d \otimes f(r)$.
   - If $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\rangle$ and $q \overset{\gamma_1}{\dashleftarrow} p_1$ is in the current automaton with weight d add a new transition $q \overset{\gamma_2}{\leftarrow} p_2$ with weight $d \otimes f(r)$.
   - If $r = \langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3\rangle$ and $q \overset{\gamma_1}{\dashleftarrow} p_1$ is in the current automaton with weight d add two new transitions $q \overset{\gamma_3}{\leftarrow} q_{p_2,\gamma_2}$ and $q_{p_2,\gamma_2} \overset{\gamma_1}{\leftarrow} p_1$ and with weights $d \otimes f(r)$ and $\bar{1}$ respectively.

Now let us consider the above on an example similar to the one from the previous section and see how the weights fit the picture.

Example 4.5 *Consider a pushdown rule $r_1 = \langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3\rangle$ and a configuration $\langle p_1, \gamma_1 s\rangle$ (where $s \in \Gamma$) that it is a successor of some configuration of interest c, namely there exists σ such that $c \overset{\sigma}{\Longrightarrow}{}^* \langle p_1, \gamma_1 s\rangle$ and thus we have in the automaton:*

$$q_f \overset{s}{\leftarrow}{}^* q \overset{\gamma_1}{\leftarrow} p_1$$

where $q \in Q$ and $q_f \in F$. But the configuration $\langle p_2, \gamma_2\gamma_3 s\rangle$ is also a successor of c, since:

$$c \overset{\sigma}{\Longrightarrow}{}^* \langle p_1, \gamma_1 s\rangle \overset{r_1}{\Longrightarrow} \langle p_2, \gamma_2\gamma_3 s\rangle$$

In order to model the fact that the stack grows, we add the additional state and have:

$$q_f \overset{s}{\xleftarrow{*}} q \overset{\gamma_3}{\longleftarrow} q_{p_2,\gamma_2} \overset{\gamma_2}{\longleftarrow} p_2$$

and the weights are: $d_1 \otimes f(r_1)$ for $q_f \overset{s}{\xleftarrow{*}} q \overset{\gamma_3}{\longleftarrow} q_{p_2,\gamma_2}$ and $\bar{1}$ for $q_{p_2,\gamma_2} \overset{\gamma_2}{\longleftarrow} p_2$. Now consider the case when there is also a rule $r_2 = \langle p_2, \gamma_2\rangle \hookrightarrow \langle p_3, \epsilon\rangle$. In this situation $\langle p_3, \gamma_3 s\rangle$ will also be a successor since we have:

$$c \overset{\sigma}{\Longrightarrow}{}^* \langle p_1, \gamma_1 s\rangle \overset{r_1}{\Longrightarrow} \langle p_2, \gamma_2\gamma_3 s\rangle \overset{r_2}{\Longrightarrow} \langle p_3, \gamma_3 s\rangle$$

This is modeled using the ϵ-transition, i.e., we have:

$$q_f \overset{s}{\xleftarrow{*}} q \overset{\gamma_3}{\longleftarrow} q_{p_2,\gamma_2} \overset{\epsilon}{\longleftarrow} p_3$$

where $q_{p_2,\gamma_2} \overset{\epsilon}{\longleftarrow} p_3$ is assigned weight $\bar{1} \otimes f(r_2)$. However, note the crucial difference here compared to Pre^* — we need to read the weights of a path in the reverse order. That is multiply the weight of the first transition at the very end. Assuming that $q_f \overset{s}{\xleftarrow{*}} q$ is just a single transition with weight d_0, we should multiply the weights in the following order $d_0 \otimes d_1 \otimes f(r_1) \otimes \bar{1} \otimes f(r_2)$.

The intuition behind reading the weights of a path in the automaton in the reverse order is that when a configuration $\langle p, \gamma_k \ldots \gamma_1\rangle$ is accepted, this means that there are transitions in the automaton such that the first one is labeled with γ_k, the second with γ_{k-1} and so on. However, when one thinks how the program would actually execute, it would build the stack from the other end, i.e before it can push γ_2 on the stack, it must push γ_1. Therefore, the weights should be multiplied in the reverse order.

4.4 Communicating pushdown systems

The *communicating pushdown systems* (CPDS) have been introduced in [10, 11] and then subsequently used in [14] to analyze concurrent C programs (where the authors managed to uncover previously unknown bug in the Bluetooth driver for the Windows operating system). CPDS in many regards are very similar to the weighted pushdown systems — both of the formalisms use PDS and annotate all the pushdown rules with some additional information. However, their purpose is obviously quite different — while WPDS can be seen as a generalization of

interprocedural dataflow analysis, the CPDS are concerned with multiple recursive processes communicating with each other using synchronization actions. And thus the annotations on the pushdown rules are often quite different: in case WPDS it is usually some dataflow state transformer, whereas in CPDS it is usually some abstraction of a synchronization action. Furthermore, the information obtained from the analysis is often used for different purposes, in WPDS it provides some information about the state of a single process, whereas in CPDS one is often interested in the reachability of a system of concurrent processes. Nevertheless, most of the principles behind CPDS and WPDS are very similar.

We we will define CPDS as originally presented in [10, 11] and we will only briefly mention some of the similarities and differences to the WPDS presentation [66, 67, 53], i.e., whenever we believe it helps with the presentation. We will spend some more time on discussing those differences in Section 4.5.

The original presentation of communicating pushdown systems [10, 11] used a slightly different definition of a pushdown system itself. We recall it below.

Definition 4.5 A pushdown system (PDS) is a four-tuple $(P, \Gamma, \Delta, \mathsf{Act})$ where P is a finite set of control locations, Γ is a finite set of stack symbols and Δ is a finite set of pushdown rules of the form $\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle p', w \rangle$, where $a \in \mathsf{Act}$, $w \in \Gamma^*$ and $|w| \leq 2$.

Note that the definition is very similar to the one of WPDS — instead of a weight function (mapping each rule to its weight) we annotate each rule with some action. In other words the difference is strictly notational and one can easily switch between the two. Therefore, we will use both of them, depending on which one is more natural and results in cleaner exposition in the given context.

Now that we can define the communicating pushdown systems.

Definition 4.6 A communicating pushdown system (CPDS) is a tuple of pushdown systems $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ over the same set of synchronization actions Act.

Note that the above definition can also be formulated using WPDS, where the weight functions have the same co-domain. However, the notation used for CPDS is often a bit more intuitive, since the set Act often corresponds to synchronization actions and one can think of a single pushdown system as generating a language of synchronization actions.

The idea of configuration of a PDS is extended to the case of CPDS in a straightforward manner. Namely, a global configuration of a CPDS is a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots, \mathcal{P}_n$. Similarly the relation $\overset{a}{\Longrightarrow}$ can be extended to global configurations:

- $g \overset{\tau}{\Longrightarrow} g'$ if there is $1 \leq i \leq n$ such that $c_i \overset{\tau}{\Longrightarrow} c'_i$ and $c'_j = c_j$ for all $j \neq i$
- $g \overset{a}{\Longrightarrow} g'$ if there are $i \neq j$ such that $c_i \overset{a}{\Longrightarrow} c'_i$ and $c_j \overset{a}{\Longrightarrow} c'_j$ ("a" is a synchronization action). Finally, for all $k \neq i \wedge k \neq j$ we have that $c'_k = c_k$.

Furthermore, it is important to recall that one of the assumptions of the CPDS as introduced in [10, 11] is that every channel is used by at most two processes (below we will see why this is important).

Now let us consider a CPDS $(\mathcal{P}_1, \ldots \mathcal{P}_n)$ and assume that we are interested in the question whether a configuration from $C'_1 \times \cdots \times C'_n$ is reachable from some configuration from $C_1 \times \cdots \times C_n$. In the following we will use $L_j = L(C_j, C'_j)$ to denote the language summarizing all paths of the process j that go from any configuration of C_j to any configuration of C'_j. The case for just two processes is not that complex — this problem is really nothing else than testing for emptiness of the intersection of the languages L_1 and L_2. The intuition behind this is that the intersection is empty only if there are no communication traces of the two processes that would match. Generalization for arbitrary number of processes requires a "shuffle" operator to modify the language of every process — we want allow for other processes to communicate (i.e., allow for communication that does not involve the current process). It can be considered as an interleaving operator that simply inserts synchronization actions of other processes into the paths of $\mathcal{P}_i$. The resulting language for process i will be denoted by $\widehat{L_i}$ and can be defined as

$$\widehat{L_i} = L_i \sqcup\!\sqcup \left(\bigcup_{k \neq i, l \neq i} \mathsf{Chan}_{k,l} \right)^*$$

where $\mathsf{Chan}_{k,l}$ are the channels used between processes k and l. This is also the reason for the requirement that every channel is used by exactly two processes [10, 11]. Note that the shuffle can also be expressed in terms of an inverse homomorphic image:

$$\widehat{L_i} = h_i^{-1} L_i$$

where h_i homomorphism is defined as

$$h_i\ (s, r, t) = \begin{cases} (s, r, t) & \text{if } r = i \vee s = i \\ \epsilon & \text{otherwise} \end{cases}$$

With the above we can finally reason about the reachability in a system of arbitrary many processes. More formally, if we have sets of global configurations $G = C_1 \times \cdots \times C_n$ and $G' = C'_1 \times \cdots \times C'_n$ and if

$$\widehat{L_1} \cap \cdots \cap \widehat{L_n} = \emptyset$$

then we can conclude that no configuration of G' is reachable from any configuration of G. However, there is still one remaining problem — the generated languages are in general context-free. To see this consider the simple example of a procedure that performs synchronization action a then either does an internal action or calls itself recursively, finally performs b and returns. Clearly the generated language is $\{a^n b^n | n \in \mathbb{N}\}$. Unfortunately, checking the emptiness of the intersection of context-free languages is undecidable. Therefore, one can use the approach of using some abstract domain for which the notion of intersection is decidable. This can be accomplished by establishing a Galois connection $(\alpha : \mathcal{P}(\mathsf{Act}^*) \to D, \gamma : D \to \mathcal{P}(\mathsf{Act}^*))$ between the complete lattice of languages over Act and some complete lattice D that also forms a semiring, i.e., $(D, \oplus_D, \otimes_D, \bar{0}_D, \bar{1}_D)$. The Galois connection can be defined in a generic manner [10, 11]:

$$\alpha(L) = \bigoplus_D \{v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \mid a_1 \cdots a_n \in L\}$$
$$\gamma(d) = \{a_1 \cdots a_n \in \mathsf{Act}^* \mid v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \sqsubseteq d\}$$

where v_a is the abstract value for synchronization action a. This allows us to move from the language of traces of a process to some abstract domain. The main motivation for this is to compute the abstraction of traces, for which the notion of "intersection" is decidable. Imposing an additional requirement:

$$\gamma(\bot) = \emptyset$$

we get the desired result:

$$\forall L_1, \ldots, L_n : \alpha(L_1) \sqcap \ldots \sqcap \alpha(L_n) = \bot \implies L_1 \cap \ldots \cap L_n = \emptyset$$

In other words if the intersection of communications is empty in the abstract domain then it is also empty for the languages of actual traces. Note that this last requirement, in our case, corresponds to demanding that $\gamma(\bar{0}_D) = \emptyset$. Apart from that, it is usually also required that $\bar{1}_D$ is the element corresponding to the $\bar{1}_L = \{\epsilon\}$, in other words:

$$\alpha(\{\epsilon\}) = v_\epsilon = \bar{1}_D$$

This corresponds to the abstraction of empty trace (e.g., a trace that corresponds to internal actions of a process) being a neutral element for $\otimes$ in the abstract

domain. This should feel quite natural — when abstracting internal actions we do want the result to be neutral to the abstraction of synchronization actions.

The procedures for computing the weighted automata for CPDS [10, 11] are quite similar to those presented in [66, 67]. The main difference is the fact that they explicitly generate the constraint system and only then solve it. The advantage of this scheme is that while the algorithms for WPDS only work for abstractions satisfying the ascending chain condition, here one can use different solvers depending on the abstraction. This obviously includes algorithms based on Kleene iteration, but also other approaches such as the recent work on Newton's method generalized to ω-continuous semirings [27], which among other things allows us to compute the least solution of equation systems over commutative abstractions that do have infinite ascending chains. We will explore this technique and its experimental implementation in Chapter 9.

Now we will present the procedures proposed for CPDS and in Section 4.5 discuss other differences when compared to the approach of WPDS. Again we assume that the initial automata do not contain any ϵ-transitions or transitions going into the initial states.

The Pre^* procedure basically proceeds exactly as in the case of PDS for computing the $\mathcal{A}_{pre^*}$, but additionally generates a number of constraints, which when solved provide the weights for the transitions of the computed automaton. Our presentation of $Post^*$ for CPDS will be slightly different than the one in [10, 11], where the weights were "multpiplied" in the same order as in the case of Pre^*. This required to introduce $(-)^R$ operator that "reversed" the weights after reading them off the $\mathcal{A}_{post^*}$ automaton. We will multiply the weights in the opposite order than in the case of Pre^*, which is closer to the approach WPDS take and also makes it possible to avoid $(-)^R$ operator.

The constraints are generated from the pushdown rules:

1. If $t = q_1 \xrightarrow{\gamma} q_2$ for some $q_1, q_2 \in Q$ was already a transition of $\mathcal{A}$ then

$$\bar{1} \sqsubseteq l(t)$$

2. For every rule $r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \gamma_2 \rangle$ and every $q \in Q$:

$$f(r) \otimes l(p_2 \xrightarrow{\gamma_2} q) \sqsubseteq l(p_1 \xrightarrow{\gamma} q)$$

3. For every rule $r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \epsilon \rangle$

$$f(r) \sqsubseteq l(p_1 \xrightarrow{\gamma} p_2)$$

4. For every rule $r = \langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$ and every $q_1 \in Q$

$$\bigoplus_{q_2 \in Q} \Big(f(r) \otimes l(p_2 \xrightarrow{\gamma_2} q_2) \otimes l(q_2 \xrightarrow{\gamma_3} q_1) \Big) \sqsubseteq l(p_1 \xrightarrow{\gamma_1} q_1)$$

The procedure for $Post^*$ is again somewhat more involved. Moreover, the approach presented in [10, 11] uses a slightly different first phase of the saturation procedure compared to WPDS.[2]

> For each pushdown rule $r \in \Delta$ of the form $r = \langle p_1, \gamma_1 \rangle \overset{a}{\hookrightarrow} \langle p_2, \gamma_2\gamma_3 \rangle$ add a new state r and a new transition $r \xleftarrow{\gamma_2} p_2$.

The constraints are generated in a similar way as in the case of Pre^*:

- If $t = q_2 \xleftarrow{\gamma} q$ is already a transition in $\mathcal{A}$ then

$$\bar{1} \sqsubseteq h(t)$$

- For every rule $r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \gamma_2 \rangle$ and every $q \in Q$

$$h'(q \xleftarrow{\gamma} p_1) \otimes f(r) \sqsubseteq h(q \xleftarrow{\gamma_2} p_2)$$

- For every rule $r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \epsilon \rangle$ and every $q \in Q$

$$h'(q \xleftarrow{\gamma} p_1) \otimes f(r) \sqsubseteq h(q \xleftarrow{\epsilon} p_2)$$

- For every rule $r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$ and every $q \in Q$

$$h'(q \xleftarrow{\gamma} p_1) \otimes f(r) \sqsubseteq h(q \xleftarrow{\gamma_3} r)$$

where the $h'(q \xleftarrow{\gamma} p)$ denotes the summary of all the runs of the form $q \overset{\gamma}{\dashleftarrow} p$:

$$h'(q \xleftarrow{\gamma} p) = h(q \xleftarrow{\gamma} p) \oplus \bigoplus \{ h(q \xleftarrow{\gamma} q') \otimes h(q' \xleftarrow{\epsilon} p) \mid q' \in Q \}$$

In both cases the result is an automaton and a set of constraints — their solution provides the weights for each of the transitions of the automaton. Note that this is slightly different than the approach taken by WPDS where the computation of weights is part of the saturation procedure itself.

[2] Again, we will discuss the difference in Section 4.5.

4.5 Brief comparison between WPDS and CPDS

The presentations of CPDS and WPDS are clearly very similar and especially in the procedures for computing the weighted automata are almost the same. However, when looking a bit more closely, one can see the approaches do differ in some details. Here we will briefly consider under what circumstances they are actually equivalent.

Consider the procedure for Pre^*. For rule of the form:

$$\langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \epsilon \rangle$$

in the CPDS approach will generate a constraint:

$$f(r) \sqsubseteq l(p_1 \xrightarrow{\gamma} p_2)$$

and in the case of WPDS we will have an operation:

$$f(r) \otimes l(p_1 \xrightarrow{\epsilon} q) \sqsubseteq l(p_1 \xrightarrow{\gamma} p_2)$$

but since the automaton does not have epsilon transitions, then $p \xrightarrow{\epsilon} q$ can only be different from $\bar{0}$ if $p = q$. So let us consider the second case and see under which circumstances it coincides with the first one. There are only two cases possible — either the $l(p \xrightarrow{\epsilon} q)$ is equal to $\bar{1}$ (when $q = p$) or otherwise it is equal to $\bar{0}$. Since both of the approaches assume a semiring structure, then we can assume that $\forall g \in G : g \otimes \bar{0} = \bar{0}$, and have two cases to consider:

- $f(r) \otimes \bar{0} = \bar{0} \sqsubseteq l(p_1 \xrightarrow{\gamma} p_2)$ so the constraint is always satisfied and does not change anything; therefore the result is the same as in the case of WPDS.
- $f(r) \otimes \bar{1} \sqsubseteq l(p_1 \xrightarrow{\gamma} p_2)$ which is, again, the same constraint as in the case of WPDS.

Which clearly makes both formulations coincide.

In case of $Post^*$ there is another slight difference in the way the weights are handled in case of rules of the form

$$r = \langle p_1, \gamma \rangle \hookrightarrow \langle p_2, \gamma_1 \gamma_2 \rangle$$

Using the approach of CPDS we will generate the following constraints:

$$f(r) \otimes \left(h(q \xleftarrow{\gamma_1} p) \oplus \bigoplus \{ h(q \xleftarrow{\gamma_1} q') \otimes h(q' \xleftarrow{\epsilon} p) \mid q' \in Q \} \right) \sqsubseteq h(q \xleftarrow{\gamma_2} q_a)$$

where q_a is the added state (it will be equal to r in case of CPDS and to q_{p_2,γ_2} in WPDS one; we will discuss this difference briefly at the end of this section). Whereas in case of WPDS we will compute (the constraint is not explicitly generated but this value will be computed by the algorithm):

$$\Big(f(r) \otimes h(q \xleftarrow{\gamma_1} p) \oplus \bigoplus\{f(r) \otimes h(q \xleftarrow{\gamma_1} q') \otimes h(q' \xleftarrow{\epsilon} p) \mid q' \in Q\}\Big) \sqsubseteq h(q \xleftarrow{\gamma_2} q_a)$$

The two inequalities are equivalent only in case the $\otimes$ operator distributes over $\oplus$. Therefore, the result will be the same for most of the original examples. However, it is known that the distributivity requirement can be relaxed to monotonicity. In this situation the results in both cases are safe, as they compute an over-approximation of the behavior of the program, but the actual result can be different depending on whether we use the approach of WPDS or CPDS.

Apart from that there is also a slight difference in the formulation of the saturation rule for push-rules. In the original formulation of CPDS [10, 11], as well as in an earlier paper [26], when considering a rule

$$r = \langle p_1, \gamma_1 \rangle \overset{a}{\hookrightarrow} \langle p_2, \gamma_2\gamma_3 \rangle$$

a state r is added (and an accompanying transition), whereas in the case of WPDS the saturation procedure adds a state q_{p_2,γ_2}. Interestingly the latter approach is also used in the work on pushdown systems by Schwoon [72]. The main consequence of adding state q_{p_2,γ_2} is that the body of the procedure will be analyzed once and its semiring value reused for different call sites (i.e., push-rules with the same "entry point" — (p_2, γ_2)). By adding a new state r the body of the procedure will be analyzed for each of the calls. Therefore, adding the state q_{p_2,γ_2} should result in less states in the constructed automaton and in computing the summarization of a procedure only once. This seems especially important since one of the main advantage of using procedure summaries is to reuse them for all call sites.

Example 4.6 *Consider the following pushdown system.*

$$\begin{aligned}
r_0 &= \langle p, m_0 \rangle \hookrightarrow \langle p, f_0 m_1 \rangle \\
r_1 &= \langle p, m_0 \rangle \hookrightarrow \langle p, f_0 m_2 \rangle \\
r_2 &= \langle p, f_0 \rangle \hookrightarrow \langle p, f_1 \rangle \\
r_3 &= \langle p, f_1 \rangle \hookrightarrow \langle p, \epsilon \rangle
\end{aligned}$$

The results of the two approaches are presented on Figures 4.1 and 4.2.

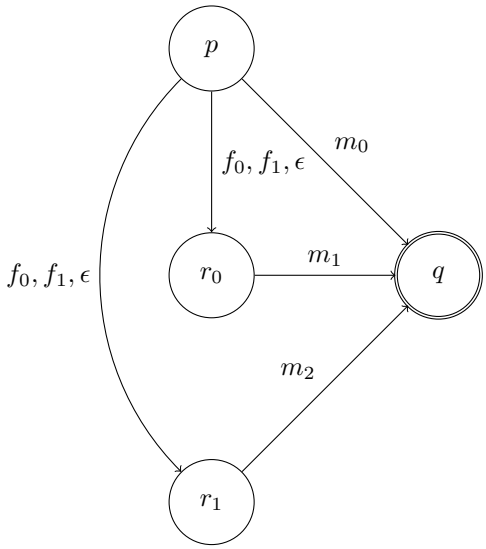


Figure 4.1: $\mathcal{A}_{post^*}$ as in CPDS papers [10, 11].

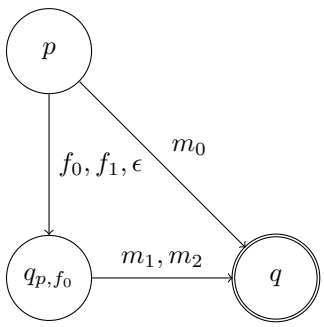


Figure 4.2: $\mathcal{A}_{post^*}$ as in WPDS papers [66, 67].

4.6 Pushdown systems and flow algebras

In the previous sections we have discussed the WPDS and CPDS, which both use semirings for the weights of pushdown rules. An interesting question is whether there is any relation between pushdown systems using semirings and the concept of flow algebras and their upper-approximations, as introduced in the Chap. 3 and [30]. One of the important developments behind CPDS is establishing the Galois connection between the sets of traces of a pushdown system and some abstraction that has computable notion of "intersection". Let us remind here its definition [10, 11]:

$$\alpha : \mathcal{P}(\mathsf{Act}^*) \to D \qquad \alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \otimes \cdots \otimes v_{a_n}$$

$$\gamma : D \to \mathcal{P}(\mathsf{Act}^*) \qquad \gamma(d) = \{a_1 \cdots a_n \in \mathsf{Act}^* \mid v_{a_1} \otimes \cdots \otimes v_{a_n} \sqsubseteq d\}$$

where v_a is the abstract value for the language $\{a\}$. Note that the $\mathcal{P}(\mathsf{Act}^*)$ forms a semiring of formal languages [23] over Act. It is defined in the obvious way:

- $L_1 \otimes_L L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$
- $\oplus_L = \cup$
- $\bar{0}_L = \emptyset$
- $\bar{1}_L = \{\epsilon\}$

Of course, the above also defines a flow algebra.

In [30] and Definition 3.5 we have defined when a flow algebra is an upper-approximation of another one. As a reminder, for a Galois connection (L, α, γ, M) we say that the flow algebra $(M, \oplus_D, \otimes_D, \bar{0}_D, \bar{1}_D)$ is an upper-approximation of $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$ if both of the below conditions are satisfied:

$$\alpha(\gamma(m_1) \otimes_L \gamma(m_2)) \sqsubseteq_D m_1 \otimes_D m_2$$
$$\alpha(\bar{1}_L) \sqsubseteq_D \bar{1}_D$$

Moreover, recall that if we have equalities (instead of less than or equal operators) in the above equations, then we say that the flow algebra M is *induced* by the Galois connection.

Note that the second condition is immediately satisfied since CPDS expects the $\bar{1}_D$ to be the abstraction of the language $\{\epsilon\}$, i.e., $\alpha(\{\epsilon\}) = \bar{1}_D$.

Now let us consider the first condition in the context of the Galois connection used for CPDS. It turns out that the generic definition of Galois connection does imply that the abstraction D is an upper-approximation of $\mathcal{P}(\mathsf{Act}^*)$ whenever D is a complete flow algebra.

Lemma 4.7 *Let $(D, \oplus_D, \otimes_D, \bar{0}_D, \bar{1}_D)$ be a complete flow algebra and*

$$(\alpha : \mathcal{P}(\mathsf{Act}^*) \to D,\ \gamma : D \to \mathcal{P}(\mathsf{Act}^*))$$

be a Galois connection as defined above (where in particular $\alpha(\bar{1}_L) = \bar{1}_D$), then $(D, \oplus_D, \otimes_D, \bar{0}_D, \bar{1}_D)$ is an upper-approximation of $(\mathcal{P}(\mathsf{Act}^*), \oplus_L, \otimes_L, \emptyset, \{\epsilon\})$.

Proof. Since the we already have that $\alpha(\bar{1}_L) = \bar{1}_D$, the only remaining thing to prove is that:

$$\alpha(\gamma(d_1) \otimes_L \gamma(d_2)) \sqsubseteq d_1 \otimes_D d_2$$

The proof uses the definition of α and γ along with the monotonicity of $\otimes_D$ as well as the fact that $\bigoplus_D$ is a least upper bound operator.

$$\begin{aligned}
&\alpha(\gamma(d_1) \otimes_L \gamma(d_2)) \\
&= [\text{ definition of } \gamma\ ] \\
&\alpha(\{a_1 \cdots a_n \in \mathsf{Act}^* \mid v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \sqsubseteq d_1\} \\
&\qquad \otimes_L \{a_1 \cdots a_n \in \mathsf{Act}^* \mid v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \sqsubseteq d_2\}) \\
&= [\text{ definition of } \otimes_L\ ] \\
&\alpha\left(\left\{ a_1 \cdots a_m a'_1 \cdots a'_n \in \mathsf{Act}^* \;\middle|\; \begin{array}{l} v_{a_1} \otimes_D \cdots \otimes_D v_{a_m} \sqsubseteq d_1 \\ v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \sqsubseteq d_2 \end{array} \right\}\right) \\
&= [\text{ definition of } \alpha\ ] \\
&\bigoplus_D \left\{ v_{a_1} \otimes_D \cdots \otimes_D v_{a_m} \otimes_D v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \;\middle|\; \begin{array}{l} v_{a_1} \otimes_D \cdots \otimes_D v_{a_m} \sqsubseteq d_1 \\ v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \sqsubseteq d_2 \end{array} \right\} \\
&\sqsubseteq [\text{ monotonicity of } \oplus_D\ ] \\
&\bigoplus_D \{v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \mid v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \sqsubseteq d_1\} \\
&\qquad \otimes_L \bigoplus_D \{v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \mid v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \sqsubseteq d_2\}) \\
&\sqsubseteq [\ D \text{ is a complete flow algebra and } \bigoplus_D \text{ is } \bigsqcup_D\ ] \\
&d_1 \otimes_D d_2
\end{aligned}$$

□

Another interesting question is when (and if) the flow algebra D can be induced by the Galois connection. It turns out that this indeed can happen if the domain D and the Galois connection satisfy some additional properties.

One of them is that α is surjective:

$$\forall d \in D : \exists L : \alpha(L) = d \tag{4.1}$$

It implies that for for all $d \in D$ we have that:

$$\bigoplus_D \{v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \mid v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \sqsubseteq d\} = d \tag{4.2}$$

because of the definition of α and the fact that $\bigoplus_D$ is a least upper bound operator on D. Then by Lemma 2.14 we have that $(\mathcal{P}(\mathsf{Act}^*), \alpha, \gamma, D)$ is a *Galois insertion*.

Moreover, if we additionally have that the abstract domain D is in fact a distributive flow algebra, then in the last steps of the proof of Lemma 4.7 we have equalities (instead of inequalities).

Lemma 4.8 *If $(D, \oplus_D, \otimes_D, \bar{0}_D, \bar{1}_D)$ is a complete, strict and affine flow algebra (i.e., it is completely distributive) and we have a Galois insertion*

$$(\alpha : \mathcal{P}(\mathsf{Act}^*) \to D, \gamma : D \to \mathcal{P}(\mathsf{Act}^*))$$

defined as above, then the flow algebra D is induced by the Galois insertion from $\mathcal{P}(\mathsf{Act}^*)$.

Proof. As before we have that $\alpha(\bar{1}_L) = \bar{1}_D$. Now we want to prove that:

$$\alpha(\gamma(d_1) \otimes_L \gamma(d_2)) = d_1 \otimes_D d_2$$

If either d_1 or d_2 is equal to $\bar{0}_D$ then the result is immediate due to the fact that $\gamma(\bar{0}_D) = \bar{0}_L$ and D is strict. Therefore, we can assume that both d_1 and d_2 are

different than $\bar{0}_D$, and we have:

$$
\begin{aligned}
&\alpha(\gamma(d_1) \otimes_L \gamma(d_2)) \\
&= [\text{ as in Lemma 4.7 }] \\
&\bigoplus_D \left\{ v_{a_1} \otimes_D \cdots \otimes_D v_{a_m} \otimes_D v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \;\middle|\; \begin{array}{l} v_{a_1} \otimes_D \cdots \otimes_D v_{a_m} \sqsubseteq d_1 \\ v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \sqsubseteq d_2 \end{array} \right\} \\
&= [\text{ } D \text{ is affine }] \\
&\bigoplus_D \{v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \mid v_{a_1} \otimes_D \cdots \otimes_D v_{a_n} \sqsubseteq d_1\} \\
&\qquad \otimes_D \bigoplus_D \{v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \mid v_{a'_1} \otimes_D \cdots \otimes_D v_{a'_n} \sqsubseteq d_2\} \\
&= [\text{ (4.2) }] \\
&d_1 \otimes_D d_2
\end{aligned}
$$

□

To sum up, the generic definition of Galois connection used for CPDS in [10, 11] is actually quite close to the notion of when a flow algebra is an upper-approximation of the flow algebra corresponding to the language of actual traces. Where *upper-approximation* is defined as in Definition 3.5, Chapter 3. Furthermore, in some of the abstractions we will often have that the Galois connection in fact *induces* the other flow algebra. This again should make intuitive sense — we have only relevant elements in the abstract domain and, in a way, preserve its structure. Also note that the original formulation of CPDS used, as one of the two main classes of abstractions, idempotent semirings satisfying the ascending chain condition. Therefore, all the abstractions in this class will also be a completely distributive flow algebras and so Lemma 4.8 applies to them.

Apart from that, note that there is also a relation to original formulation of WPDS. It is not as pronounced as in the case of CPDS where the original definition also specified the definition of the Galois connection. Nevertheless, as already mentioned, the WPDS model is closely related to the functional approach [73] to interprocedural analysis. In other words one can think of computing summarizations of procedures and statements and composing them together. This is very close to the idea behind the [30], therefore we do expect that the concepts of upper-approximation and induced flow algebras are useful also in this context.

Chapter 5

Pushdown systems for monotone frameworks

As already mentioned, the weighted and communicating pushdown systems are a powerful technique of modeling recursive programs. Furthermore, by using semirings as the basic requirement for abstract domains, they allow a variety of different analyses. However, due to the fact that semirings require the annihilation property, they do not directly admit some of the classical analyses that are not strict (e.g., analyses based on bit vector frameworks). In this chapter we will show that it is possible to use flow algebras in the context of weighted/communicating pushdown systems and thus dispose of the annihilation requirement.

The structure of the chapter is as follows. First, we introduce and motivate our work in Section 5.1. Then in Section 5.2 we recall and introduce the concepts necessary for the remainder of the chapter (e.g., monotone frameworks). Section 5.3 presents the basic definitions, while Section 5.4 describes our algorithms and provides some intuition behind them. In Section 5.5 we present the soundness result for both the forward and backward reachability. Similarly, Section 5.6 describes the completeness results for both of them. Finally, we discuss our approach and results in Section 5.7 and conclude in Section 5.8.

Acknowledgement: This chapter is based on [77].

5.1 Introduction

5.1.1 Monotone frameworks

Monotone frameworks [45] is a unifying approach to static analysis of programs. It creates a generic foundation for specifying various analyses and by imposing very modest requirements it can accommodate a wide range of analyses, including the bit vector frameworks as well as more complex ones such as constant propagation. However, the original formulation was focused on the intraprocedural setting and did not discuss the interprocedural one.

As we have already discussed in Chapter 4, interprocedural analysis has always been an interesting challenge for static analysis. Two of the main reasons for that are the unbounded stack and recursive (or mutually recursive) procedures. Moreover, only some paths in the interprocedural flow graph are valid — the call and returns should always match. All of this opens up many possibilities for various trade-offs, such as taking into account or ignoring the calling context. Recall that in their seminal work Sharir and Pnueli [73] presented two approaches allowing for precise interprocedural analysis. One of them, known as the *call-strings* approach, is based on "tagging" the analysis information with the current call stack. Obviously the length of call-strings should be limited to some threshold in order to ensure the termination of the analysis. However, in this chapter we will be more interested in the other presented approach. It is called the *functional* approach and is based on the idea of computing the summarizations of procedures, i.e., establishing the relationship between the inputs and the outputs of the blocks of the program and procedures (composing the results for the blocks). A similar idea, from the abstract interpretation perspective, was explored in [18], which considered predicate transformers as the basis for the analysis and also involved constructing systems of functional equations.

5.1.2 Pushdown systems

Pushdown systems [9, 26, 72] (introduced in Chapter 4) are one of the more recently proposed approaches to interprocedural analysis. One of the main underlying ideas behind them is to use a construction similar to pushdown automata in order to model the use of a stack by a program. It is also important to emphasize that an interesting advantage of this approach is the ability to compute the (possibly) infinite sets of predecessor and successor configurations for a given program and some initial configurations. Since the pushdown systems can only handle programs with finite abstractions, they have been extended

with semiring weights/annotations in weighted pushdown systems [66, 67, 53] and communicating pushdown systems [10, 11]. The extensions proposed in both of these approaches are actually very close (cf. Chapter 4), although the former focuses on dataflow analysis and generalizing the functional approach to interprocedural analysis, while the latter on the abstractions of language generated by synchronization actions in a concurrent setting. Pushdown systems have been used for verification purposes in many different projects and contexts. The examples include the Moped [72] and jMoped [75] model checkers that extensively use pushdown systems or Codesurfer [4] that takes advantage of weighted pushdown systems.

5.1.3 Motivation and contributions

Both the WPDS and CPDS use semirings for analysis purposes and thus exclude many classical approaches, such as bit vector frameworks where the transfer functions are not strict. In this chapter we are bringing the pushdown systems based analysis closer to the monotone frameworks. To achieve that we use the concept of flow algebra [30] (introduced in Chapter 3) that is a structure similar a semiring, but with less requirements imposed on the $\otimes$ operator. In particular recall that we do not impose the annihilation requirement, nor the distributivity. This allows us to present examples of classical analyses that thanks to our extensions are admitted by the framework, and did not directly fit into the previous semiring-based approaches.[1] Since the existing algorithms are based on the assumption of working with semiring structure, we develop our slightly different algorithms that allow us to relax the requirements. Then we go on to establish the soundness result, i.e., the analysis result safely over-approximates the join over all valid paths of the pushdown system. Furthermore, we also prove the completeness of the analysis, that is, provided that the flow algebra satisfies certain additional properties the result of the analysis will coincide with the join over all valid paths.

5.2 Monotone frameworks, semirings and flow algebras

In this section we will present the basic definitions that will be used throughout the rest of the chapter. We will start with recalling the classical approach to

[1] Although it is possible to sidestep this problem by introducing an "artificial" annihilator to the semiring.

static analysis known as monotone frameworks [45, 59]. Here we present a slightly more convenient (in the context of this chapter) definition of monotone framework.

Definition 5.1 A complete monotone framework is a tuple

$$(L, \bigsqcup, \mathcal{F}, \circ, id, (f_l)_{l \in L})$$

where L is a complete lattice, $\bigsqcup$ is its least upper bound operator. We use $\mathcal{F}$ to denote a monotone function space on L, i.e., a set of monotone functions that contains the identity function and is closed under function composition. Finally, $\circ$ is function composition, id is the identity function and $f_l = \lambda l'.l$ for every $l \in L$.

We will also discuss bit vector frameworks, which are a special case of monotone frameworks. The lattice used is $L = \mathcal{P}(D)$ for some finite set D, the ordering is either $\subseteq$ or $\supseteq$ and the least upper bound is either $\cup$ or $\cap$ and the monotone and distributive function space is defined as

$$\{f : \mathcal{P}(D) \to \mathcal{P}(D) \mid \exists Y_f^1, Y_f^2 \subseteq D : \forall Y \subseteq D : f(Y) = (Y \cap Y_f^1) \cup Y_f^2\}$$

One of the main reasons for distinguishing them is the fact that they can be implemented very efficiently using bit vectors and include common analyses such as live variables, available expressions, reaching definitions, etc. [59].

As already mentioned we will use the notion of a flow algebra, which is similar to idempotent semirings, but less restrictive. The main difference is that flow algebras do not require the distributivity and annihilation properties. Instead we replaced the first one with a monotonicity requirement and dispensed with the second one. It was previously defined in Definition 3.1 (Section 3.2). For the reader's convenience we recall its basic properties. A flow algebra is a structure of the form $(F, \oplus, \otimes, \bar{0}, \bar{1})$ such that:

- $(F, \oplus, \bar{0})$ is an idempotent and commutative monoid
- $(F, \otimes, \bar{1})$ is a monoid
- $\otimes$ is monotonic in both arguments, that is:

$$f_1 \sqsubseteq f_2 \;\Rightarrow\; f_1 \otimes f \sqsubseteq f_2 \otimes f$$
$$f_1 \sqsubseteq f_2 \;\Rightarrow\; f \otimes f_1 \sqsubseteq f \otimes f_2$$

where $f_1 \sqsubseteq f_2$ if and only if $f_1 \oplus f_2 = f_2$. Similarly we have defined a distributive ($\otimes$ distributes over $\oplus$) and strict flow algebra ($\bar{0}$ is an annihilator for

$\otimes$) in Definition 3.2. Recall that every idempotent semiring is also a strict and distributive flow algebra. Finally, we also introduced a complete flow algebra in Definition 3.4.

One of the motivations of flow algebras is that the classical bit vector frameworks [59] are not strict; hence they are not directly expressible using idempotent semirings. Therefore, from this perspective the flow algebras are closer to Monotone Frameworks, and other classical static analyses. Restricting our attention to semirings rather than flow algebras would mean restricting our attention to strict and distributive frameworks.

Let us emphasize the connection between the flow algebras and the monotone frameworks. As defined above a complete monotone framework is

$$(L, \bigsqcup, \mathcal{F}, \circ, id, (f_l)_{l \in L})$$

Note that this immediately gives us a flow algebra by taking

$$(F, \overset{\circ}{\bigsqcup}, \fatsemi, f_\perp, id)$$

where $\overset{\circ}{\bigsqcup} Y = \lambda l. \bigsqcup_{f \in Y} f(l)$ and $f \fatsemi g = g \circ f$.

5.3 Pushdown systems

As presented in Section 4.3 and 4.4 the weighted and communicating pushdown systems use slightly different definitions of pushdown systems as well as use a bit different notation. Therefore, to avoid any confusion, we will present below the definitions that are applicable to this chapter. We will mostly follow the notation used for WPDS (as already noted, even though CPDS approach is slightly different, the basic ideas are almost the same in both cases).

Definition 5.2 A pushdown system is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is a finite set of control locations, Γ is a finite set of stack symbols and Δ is a finite set of pushdown rules of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$, where $w \in \Gamma^*$ and $|w| \leq 2$.

The pushdown systems themselves require that the sets P and Γ are finite, which makes it impossible to use infinite abstractions. Recall that in order to allow using such abstractions with pushdown systems the papers [66, 67, 10, 11] equipped every pushdown rule with a semiring value.

Definition 5.3 A weighted pushdown system a tuple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$, where $\mathcal{P}$ is a pushdown system, $\mathcal{S} = (S, \oplus, \otimes, \bar{0}, \bar{1})$ is an idempotent flow algebra and $f : \Delta \to S$ maps pushdown rules to the elements of S.

The main difference when compared to the original definition is that we require a flow algebra instead of bounded and idempotent semiring.[2]

Recall that we define the weight of a sequence of pushdown rules as follows. Let $\sigma = [r_1, \ldots, r_n] \in \Delta^*$ be such a sequence, then we define $v(\sigma) = f(r_1) \otimes \cdots \otimes f(r_n)$.

Since the pushdown rules are equipped with weights, the Pre^* and $Post^*$ algorithms need to be adapted to return a weighted NFAs. Thus, apart from making it possible to answer reachability queries, they also provide additional dataflow information for the given configuration. In other words, we can not only ask whether a configuration is a successor or predecessor but also what is the flow algebra value of getting from that configuration (Pre^*) or to that configuration ($Post^*$). More formally, we additionally compute the following information:

- in case of predecessors of some regular set of configurations C (i.e., if c_1 is a predecessor of some configuration in C)

  $$\delta(c_1) = \bigoplus\{v(\sigma) \mid c_1 \overset{\sigma}{\Longrightarrow}{}^* c_2, c_2 \in C\}$$

  is the flow algebra value of all the paths going from configuration $c_1 = \langle p, s\rangle$ ($s \in \Gamma^*$) to any configuration in C. It can be obtained by simulating $\mathcal{A}_{pre^*}$ from state p with input s multiplying the weights of the transitions in the same order as they are taken.

- in case of successors of some regular set of configurations C (i.e., if c_1 is a successor of some configuration in C)

  $$\delta(c_1) = \bigoplus\{v(\sigma) \mid c_2 \overset{\sigma}{\Longrightarrow}{}^* c_1, c_2 \in C\}$$

  is the flow algebra value of all the paths going from any configuration in C to $c_1 = \langle p, s\rangle$ ($s \in \Gamma^*$). It can be obtained by simulating $\mathcal{A}_{post^*}$ from state p with input s multiplying the weights of the transitions in the reverse order as they are taken.

Note that in both cases we only want to calculate the value for a predecessor or successor, thus the sets of paths are never empty.[3]

[2] Bounded is used here to mean that it contains no infinite ascending chains [66, 67].
[3] This is one of the changes compared to the original formulations of WPDS and CPDS.

5.4 Algorithms

As already mentioned, WPDS and CPDS are assuming that the abstract domain forms a semiring structure. This immediately excludes standard analyses based on monotone framework or bit vector framework. Fortunately we will show that it is possible to formulate algorithms for Pre^* and $Post^*$ that do not need this assumption. We achieve that by generating the constraints during the saturation procedures that create the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata (in WPDS no constraints are generated and the weights are calculated directly, in CPDS constraints are generated independently of the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ construction). We will use $\mathcal{A}^{\mathcal{C}}_{pre^*}$ and $\mathcal{A}^{\mathcal{C}}_{post^*}$ to denote the automata with the associated set of constraints $\mathcal{C}$. The rest of the section will introduce the algorithms and in the subsequent sections we will discuss their soundness and completeness. In this way we believe that we can present the minimum requirements that are necessary for interprocedural analysis based on pushdown systems.

5.4.1 Algorithm for Pre^*

The procedure introduced in this section is quite similar to the one from [10, 11] as it generates explicit constraints. However, it does it during the automaton computation not separately. In this respect it is somewhat similar to the procedure from [66, 67] that computes both the weights and the automaton at the same time. Also, note that there is no difference with respect to how the new transitions are added to the automaton. Therefore, we are able to reuse the standard results with respect to the automaton itself (i.e., excluding the weights).

The algorithm is as follows. First, for every transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}$ we add a constraint

$$\bar{1} \sqsubseteq l(q \xrightarrow{\gamma} q')$$

(we use $l(-)$ in the constraints to denote the weight of the given transition) Then we perform the saturation procedure on $\mathcal{A}$ along with the generation of constraints that are added to $\mathcal{C}$. For every pushdown rule r in Δ:

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle$ we add a transition

  $$p \xrightarrow{\gamma} p'$$

  along with the following constraint

  $$f(r) \sqsubseteq l(p \xrightarrow{\gamma} p')$$

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a transition $p' \xrightarrow{\gamma'} q$ in the current automaton, we add a transition

$$p \xrightarrow{\gamma} q$$

along with the following constraint

$$f(r) \otimes l(p' \xrightarrow{\gamma'} q) \sqsubseteq l(p \xrightarrow{\gamma} q)$$

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there is a path $p' \xrightarrow{\gamma'} q' \xrightarrow{\gamma''} q$ (for some q') in the current automaton, we add a transition

$$p \xrightarrow{\gamma} q$$

along with the following constraint

$$f(r) \otimes l(p' \xrightarrow{\gamma'} q') \otimes l(q' \xrightarrow{\gamma''} q) \sqsubseteq l(p \xrightarrow{\gamma} q)$$

We stop once we cannot add any new constraints or transitions. And since the number of possible transitions and constraints is finite, the procedure will always terminate.

5.4.2 Algorithm for *Post**

As in the case of *Pre** algorithm, we only change the way the constraints are generated, and not how new transitions are added to the automaton. Recall that we require the initial automaton $\mathcal{A}$ to have no transitions going into the initial states nor any ϵ-transitions. We will use the reverse arrow notation for the transitions of the automata as in the Chapter 4, i.e., we will write $q \xleftarrow{\gamma} p$ for the transition earlier denoted by $p \xrightarrow{\gamma} q$. Furthermore, recall that ϵ-transitions added by the algorithm always originate in an initial state and go only to some non-initial state. Thus we use $\overset{\gamma}{\leftarrow\!\!-}$ to denote $(\xleftarrow{\gamma} \circ \xleftarrow{\epsilon}) \cup \xleftarrow{\gamma}$ and define h^ϵ as

$$h^\epsilon(\rho) = \begin{cases} h(q \xleftarrow{\gamma} p) & \text{if } \rho = q \xleftarrow{\gamma} p \\ h(q \xleftarrow{\gamma} q') \otimes h(q' \xleftarrow{\epsilon} p) & \text{if } \rho = q \xleftarrow{\gamma} q' \xleftarrow{\epsilon} p \end{cases}$$

The algorithm is as follows. First, for every transition $q' \xleftarrow{\gamma} q$ in $\mathcal{A}$ we add a constraint

$$\bar{1} \sqsubseteq h(q' \xleftarrow{\gamma} q)$$

Then for all pushdown rules of the form $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ we add a new state $q_{p',\gamma'}$ to the automaton. Finally, for every pushdown rule r in Δ:

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle$ and there is a path $\rho = q \overset{\gamma}{\dashleftarrow} p$ then add a transition

$$q \overset{\epsilon}{\leftarrow} p'$$

along with the following constraint

$$h^{\epsilon}(q \underset{\rho}{\overset{\gamma}{\dashleftarrow}} p) \otimes f(r) \sqsubseteq h(q \overset{\epsilon}{\leftarrow} p')$$

Note that this transition (and its weight) takes care of the return from a procedure.

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a path $\rho = q \overset{\gamma}{\dashleftarrow} p$ then add a transition

$$q \overset{\gamma'}{\longleftarrow} p'$$

along with the following constraint

$$h^{\epsilon}(q \underset{\rho}{\overset{\gamma}{\dashleftarrow}} p) \otimes f(r) \sqsubseteq h(q \overset{\gamma'}{\longleftarrow} p')$$

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there is a path $\rho = q \overset{\gamma}{\dashleftarrow} p$ then add transitions

$$q \overset{\gamma''}{\longleftarrow} q_{p',\gamma'} \qquad q_{p',\gamma'} \overset{\gamma'}{\longleftarrow} p'$$

along with the following constraints

$$\bar{1} \sqsubseteq h(q_{p',\gamma'} \overset{\gamma'}{\longleftarrow} p')$$

$$h^{\epsilon}(q \underset{\rho}{\overset{\gamma}{\dashleftarrow}} p) \otimes f(r) \sqsubseteq h(q \overset{\gamma''}{\longleftarrow} q_{p',\gamma'})$$

Note that this transition $q \overset{\gamma''}{\longleftarrow} q_{p',\gamma'}$ (and its weight) takes care of the procedure call.

Again, as in the case of Pre^* we stop once we cannot add any new constraints or transitions. And since the number of possible transitions and constraints is finite, the procedure will always terminate (note that we add some new states only at the beginning of the procedure and not in the saturation phase).

5.5 Soundness

In this section we will discuss and present the main results regarding the soundness of our algorithms. Since one of the goals of our formulation of the algorithms

is to make the requirements imposed on the abstract domain explicit and precise, we take a particular approach to the soundness proofs. We do not discuss how the generated constraints can be solved (and if they can be solved at all). Instead we assume that some solution to those constraints is available and show that it is a safe over-approximation of the join over all valid paths.

Apart from that, separating the requirements necessary to solve the constraints from the soundness result gives us the flexibility to easily accommodate different techniques of solving the constraints. One can use the usual Kleene iteration as well as, e.g., the Newton's method that was recently generalized to ω-continuous semirings [27] (we will describe an experimental solver implementing this technique in Chapter 9). Furthermore, it also makes it clear that techniques such as widening can be used for domains that contain infinite ascending chains but do not satisfy the requirements of Newton's method.

5.5.1 *Pre**

We will start with some intuition about how the pushdown system $\mathcal{P}$ and the automaton $\mathcal{A}$ fit together. Observe that if a configuration is backward reachable from C, there exists a sequence of pushdown rules in Δ such that the resulting configuration is accepted by $\mathcal{A}$. Therefore, we can intuitively think about this system as a one big pushdown system $\mathcal{PA} = (P, \Gamma, \Delta_{pre})$, where

$$\Delta_{pre} = \Delta \cup \{\langle q, \gamma\rangle \hookrightarrow \langle q', \epsilon\rangle \mid q \xrightarrow{\gamma} q' \in \rightarrow\}$$

With each added pushdown rule we associate the weight $\bar{1}$. This system works by first acting like $\mathcal{P}$ and then, at some point, switching to simulating $\mathcal{A}$ (with the added pushdown rules). Note that once $\mathcal{PA}$ starts using the added pushdown rules, it cannot use the ones of $\mathcal{P}$. This is because rules in $\mathcal{P}$ correspond to the initial states of $\mathcal{A}$ and since it does not have any transitions going to initial states, then the first used rule from $\Delta_{pre} \setminus \Delta$ will go to some non-initial state. Thus no pushdown rule of $\mathcal{P}$ will be applicable.

This is useful because it allows us to look at the problem of predecessors of C from a slightly different angle. Let us consider the automaton $\mathcal{A}_{pre^*}$, we say that a configuration c_p is a predecessor of some configuration $c \in C$ if there is a sequence $\sigma \in \Delta^*$ of pushdown rules such that $c_p \overset{\sigma}{\Longrightarrow}{}^* c$. But since c is recognized by $\mathcal{A}$ then there is a sequence $\sigma' \in \Delta^*_{pre}$ such that $c \overset{\sigma'}{\Longrightarrow}{}^* \langle q_f, \epsilon\rangle$ for some final state q_f. Therefore, an alternative way to define a predecessor is to say that a configuration c_p is a predecessor of some configuration c in C if there is a sequence $\sigma_p \in \Delta^*_{pre}$ of pushdown rules such that $c_p \overset{\sigma_p}{\Longrightarrow}{}^* \langle q_f, \epsilon\rangle$ for some

state $q_f \in F$. Moreover, since we have that each of the added rules has weight $\bar{1}$ then $v(\sigma) = v(\sigma_p)$.

In the following sections the solution to the constraints will be denoted as λ (i.e., maps each transition to its weight). Its generalization to paths λ^* is inductively defined as follows:

$$\lambda^*(\rho) = \begin{cases} \lambda(q \xrightarrow{\gamma} q') & \text{if} \quad \rho = q \xrightarrow{\gamma} q' \\ \\ \lambda(q \xrightarrow{\gamma} q'') \otimes \lambda^*(\rho') & \text{if} \quad \rho = q \xrightarrow{\gamma} \overbrace{q'' \xrightarrow{s'}{}^* q'}^{\rho'} \end{cases}$$

Now we are ready to prove that a solution to the constraints generated by our saturation procedure is sound.

Theorem 5.4 *Consider an automaton $\mathcal{A}$ and its corresponding $\mathcal{A}^{\mathcal{C}}_{pre^*}$ generated by the saturation procedure. Let us assume that we have a solution λ to the set of constraints $\mathcal{C}$. Then for each pair (p, s) such that $\langle p, s\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_f, \epsilon\rangle$ (where $\sigma \in \Delta^*_{pre}$ and $q_f \in F$), we have $v(\sigma) \sqsubseteq \lambda^*(\rho)$ where $\rho = p \xrightarrow{s}{}^* q_f$ is in $\mathcal{A}_{pre^*}$.*

Proof. The proof is available in App. B.1.1.

5.5.2 *Post**

As previously we can think about this system as a one big pushdown system. However, this time such a system would first simulate the reverse of $\mathcal{A}$, i.e., instead of accepting some configuration, it generates one; and only then continue by running the pushdown system itself. Let us denote such a system as $\mathcal{A}^R\mathcal{P} = (P, \Gamma, \Delta_{post})$, where Δ_{post} is defined as follows.

- For every $q' \xleftarrow{\gamma} q$ in $\mathcal{A}$ we have a rule $r = \langle q', \epsilon\rangle \hookrightarrow \langle q, \gamma\rangle$ in Δ_{post} such that $f(r) = \bar{1}$.
- All other rules of Δ are included in Δ_{post}.

Let us consider the automaton $\mathcal{A}_{post^*}$. We say that a configuration c' is a successor of some configuration c in C if there is a sequence $\sigma \in \Delta^*$ of pushdown rules such that $c \overset{\sigma}{\Longrightarrow}{}^* c'$. But since c is recognized by $\mathcal{A}$ then there is a sequence $\sigma' \in \Delta^*_{post}$ such that $\langle q_f, \epsilon\rangle \overset{\sigma'}{\Longrightarrow}{}^* c$ for some final state q_f. Therefore, an alternative way to define a successor is to say that a configuration c' is a successor

of some configuration $c \in C$ if there is a sequence $\sigma_p \in \Delta^*_{post}$ of pushdown rules such that $\langle q_f, \epsilon \rangle \overset{\sigma_p}{\Longrightarrow}{}^* c'$ for some state $q_f \in F$. Moreover, since we have that each of the added rules has weight $\bar{1}$ then $v(\sigma) = v(\sigma_p)$.

Similarly as in the case of Pre^*, we define λ^*_R in the following way:

$$\lambda^*_R(\rho) = \begin{cases} \lambda(q \overset{\gamma}{\leftarrow} q') & \text{if} \quad \rho = q' \overset{\gamma}{\leftarrow} q \\ \\ \lambda^*_R(\rho') \otimes \lambda(q \overset{\gamma}{\leftarrow} q'') & \text{if} \quad \rho = \overbrace{q' \,{}^*\!\overset{s'}{\leftarrow} q''}^{\rho'} \overset{\gamma}{\leftarrow} q \end{cases}$$

As already mentioned we multiply the weight in the reverse order compared to the order of transitions in the given path.

Theorem 5.5 *Consider an automaton $\mathcal{A}$ and its corresponding $\mathcal{A}^{\mathcal{C}}_{post^*}$ generated by the saturation procedure. Let us assume that we have a solution λ to the set of constraints $\mathcal{C}$. Then for each pair (p, s) such that $\langle q_f, \epsilon \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p, s \rangle$ (where $\sigma \in \Delta^*_{post}$ and $q_f \in F$), we have $v(\sigma) \sqsubseteq \lambda^*_R(\rho)$ where $\rho = q_f \,{}^*\!\overset{s}{\leftarrow} p$ is in $\mathcal{A}^{\mathcal{C}}_{post^*}$.*

Proof. The proof is available in App. B.1.2.

5.6 Completeness

In this section we will prove the completeness of our procedure, i.e., we will show that provided the abstract domain satisfies certain conditions, the solution to the generated constraints will coincide with the join over all valid paths. The presentation of the results (and their proofs) is quite different than in the case of soundness. This is mainly due to the additional complexity of the proofs as well as some additional restrictions that must be imposed. Throughout the whole section we assume that the flow algebra is both *complete* and *affine*. In other words we have least upper bounds of arbitrary sets and $\otimes$ distributes over sums of all non-empty sets.

Before we present the main results for each of the two algorithms, let us first establish that the solution to the generated constraints can be obtained by Kleene iteration. To achieve that we will define a function that represents the constraints and show that it is continuous. Let us recall that all generated constraints are of similar form: the right-hand side is a variable and the left-hand side is a finite expression mentioning at most two variables. The finite expressions are constructed using $\oplus$ and $\otimes$ which are themselves affine and hence continuous.

For clarity let $\mathcal{C}_t \subseteq \mathcal{C}$ denote the finite set of the constraints that have the variable t on the right-hand side. Recall that each variable corresponds to a transition in an automaton. Similarly we will use $lhs_m(c)$ $(c \in \mathcal{C})$ to denote the interpretation of the left-hand side of the constraint c under the assignment m.

What we want to compute is a mapping m that is a fixed point of:

$$F : (\delta \to D) \to (\delta \to D)$$
$$F(m)t = \bigoplus_{c \in \mathcal{C}_t} lhs_m(c)$$

where δ is the set of all transitions.

Lemma 5.6 *F is continuous, i.e., for any non-empty chain Y:*

$$F(\bigsqcup Y) = \bigsqcup_{m \in Y} F(m)$$

Proof. The proof is available in App. B.2.

It follows that $\bigsqcup\{F^n(\bot) \mid n \in \mathbb{N}\}$ is the least solution to our constraint system.

5.6.1 *Pre**

We will first establish a lemma showing that every transition in the $\mathcal{A}_{pre^*}$ automaton has at least one corresponding path in the $\mathcal{PA}$. This will be useful in subsequent proofs where we need the fact that certain sets of $\mathcal{PA}$ paths are not empty.

Lemma 5.7 *For every transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}_{pre^*}$ there exists a sequence $\sigma \in \Delta_{pre}$ such that $\langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle$.*

Proof. The proof is available in App. B.3.1.

First we will establish the essential result for a single transition of the created automaton.

Lemma 5.8 *Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$, where $\mathcal{F}$ is affine, and an automaton $\mathcal{A}^{\mathcal{C}}_{pre^*}$ created by the saturation procedure. Moreover,*

let λ be the least solution to the set of constraints $\mathcal{C}$. For every transition $q \xrightarrow{\gamma} q'$ in this automaton we have that

$$\lambda(q \xrightarrow{\gamma} q') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \gamma\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon\rangle, \sigma \in \Delta^*_{pre}\}$$

Proof. The proof is available in App. B.3.2.

This is also the place that we have used the fact that the solution is equal to the least upper bound of the ascending Kleene sequence.

And now we can generalize the above to the case of a path in the automaton.

Lemma 5.9 *Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$, where $\mathcal{F}$ is affine, and a $\mathcal{A}^{\mathcal{C}}_{pre^*}$ automaton created by the saturation procedure. Moreover, let λ be the least solution to the set of constraints $\mathcal{C}$. For every path $\rho = q \xrightarrow{s}{}^* q'$ in this automaton we have that*

$$\lambda^*(q \xrightarrow[\rho]{s}{}^* q') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, s\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon\rangle, \sigma \in \Delta^*_{pre}\}$$

Proof. The proof is available in App. B.3.3.

And finally, using both the Theorem 5.4 and the above lemma, we can formulate the main result.

Theorem 5.10 *Consider an automaton $\mathcal{A}^{\mathcal{C}}_{pre^*}$ constructed by the saturation procedure and let λ be the least solution to the set of its constraints $\mathcal{C}$. If the flow algebra is affine then for every path $\rho = p \xrightarrow{s}{}^* q_f$ where $q_f \in F$ we have that*

$$\lambda^*(p \xrightarrow[\rho]{s}{}^* q_f) = \bigoplus\{v(\sigma) \mid \langle p, s\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_f, \epsilon\rangle, \sigma \in \Delta^*_{pre}\}$$

Proof. The proof is available in App. B.3.4.

5.6.2 *Post**

Consider a pushdown system $\mathcal{P}$ with pushdown rules Δ and a regular set of configurations C with an automaton $\mathcal{A}$ that accepts C. First let us define a small modification of the pushdown rules Δ. Each rule r of the form

$$\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma_1\gamma_2\rangle$$

can be "split" into two rules r_1 and r_2:

$$r_1 = \langle p, \gamma \rangle \hookrightarrow \langle q_{p',\gamma_1}, \gamma_2 \rangle$$
$$r_2 = \langle q_{p',\gamma_1}, \epsilon \rangle \hookrightarrow \langle p', \gamma_1 \rangle$$

with weights $f(r_1) = f(r)$ and $f(r_2) = \bar{1}$. Note that the second rule is not really a pushdown rule as defined earlier. Fortunately, all we need to do, is to redefine $\Longrightarrow$ in the following way:

$$\text{if} \quad r = \langle q, \gamma \rangle \hookrightarrow \langle q', w \rangle \qquad \text{then} \qquad \forall w' \in \Gamma^* : \langle q, \gamma s \rangle \Longrightarrow \langle q', ws \rangle$$
$$\text{if} \quad r = \langle q, \epsilon \rangle \hookrightarrow \langle q', \gamma \rangle \qquad \text{then} \qquad \forall w' \in \Gamma^* : \langle q, s \rangle \Longrightarrow \langle q', \gamma s \rangle$$

This does not change the pushdown system in any way. Since we add a fresh state, there is no danger of changing any paths except for the ones we intend to. Moreover, the weight remains the same ($\bar{1}$ is neutral element for $\otimes$, so $f(r_1) \otimes f(r_2) = f(r)$).

Therefore, in place of Δ_{post} we will use $\Delta_{post\text{-}2}$, which is defined as follows:

- For every $q' \xleftarrow{\gamma} q$ in $\mathcal{A}$ we have a rule $r = \langle q', \epsilon \rangle \hookrightarrow \langle q, \gamma \rangle$ in $\Delta_{post\text{-}2}$ such that $f(r) = \bar{1}$.
- For every $r \in \Delta$ of the form $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma_1 \gamma_2 \rangle$ there are r_1 and r_2 in $\Delta_{post\text{-}2}$ as described above.
- All other rules of Δ are included in $\Delta_{post\text{-}2}$ without any modification.

So compared to Δ_{post} the only difference is that we split the push-rules into two separate rules. At the same time we do not change the behavior of the system in any way.

This allows us to prove the following lemma, which is used in subsequent proofs.

Lemma 5.11 *For every transition $q' \xleftarrow{\gamma_\epsilon} q$ ($\gamma_\epsilon \in \Gamma \cup \{\epsilon\}$) in $\mathcal{A}_{post^*}$ there exists a sequence σ of pushdown rules in $\Delta_{post\text{-}2}$ such that $\langle q', \epsilon \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, \gamma_\epsilon \rangle$.*

Proof. The proof is available in App. B.3.5.

Again, as in the case of Pre^* we first establish the result for a single transition in the automaton.

Lemma 5.12 *Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$, where $\mathcal{F}$ is affine, and an automaton $\mathcal{A}^{\mathcal{C}}_{post^*}$ created by the saturation procedure. Moreover,*

let λ be the least solution to the set of constraints $\mathcal{C}$. For every transition $q' \xleftarrow{\gamma_\epsilon} q$ ($\gamma_\epsilon \in \Gamma \cup \{\epsilon\}$) in this automaton we have that

$$\lambda(q' \xleftarrow{\gamma_\epsilon} q) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, \gamma_\epsilon\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

Proof. The proof is available in App. B.3.6.

And again, as in the case of Pre^*, this is the place that we have used the fact that the solution is equal to the least upper bound of the ascending Kleene sequence.

Now we can generalize the obtained result for the paths in the automaton.

Lemma 5.13 *Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$, where $\mathcal{F}$ is affine, and a $\mathcal{A}^{\mathcal{C}}_{post^*}$ automaton created by the saturation procedure. Moreover, let λ be the least solution to the set of constraints $\mathcal{C}$. For every path $\rho = q' \xleftarrow{s} q$ ($s \in \Gamma^*$) in this automaton we have that*

$$\lambda^*_R(q' \overset{s}{\underset{\rho}{\leftarrow}}{}^* q) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, s\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

Proof. The proof is available in App. B.3.7.

And finally using both the soundness Theorem 5.5 and the above lemma, we can establish the main result.

Theorem 5.14 *Consider an automaton $\mathcal{A}^{\mathcal{C}}_{post^*}$ constructed by the saturation procedure and let λ be the least solution to the set of its constraints $\mathcal{C}$. If the flow algebra is affine then for every path $\rho = q_f \overset{s}{\leftarrow}{}^* p$ where $q_f \in F$ we have that*

$$\lambda^*_R(q_f \overset{s}{\underset{\rho}{\leftarrow}}{}^* p) = \bigoplus\{v(\sigma) \mid \langle q_f, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p, s\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

Proof. The proof is available in App. B.3.8.

5.7 Discussion

In this section we would like to discuss the relation of our development to the area of interprocedural analysis, as well as the challenges and advantages of the approach.

To put our approach into perspective, it is useful to emphasize that it is a generalization of the functional approach to interprocedural analysis by Sharir and Pnueli [73]. In both of these approaches the underlying idea is to compute the summarizations of actions and by composing them obtain the summarizations of procedures. The generality of weighted pushdown systems stems from the fact that they make it possible to obtain the analysis information for a specific calling context or even families of calling contexts. In other words one can perform queries of weighted $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata, to get the summarization of all the paths between the initial set of configurations and a given stack or even a regular set of stacks. Applying the summarization to some initial analysis information, we can obtain the desired result. This is possible due to the way the algorithms for pushdown systems construct the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata and generate the constraints whose solution provides us with the weights of all the transitions in those automata.

One of the most significant advantages of using summarizations is the fact that each procedure can be analyzed only once and the result can be used at all the call sites. In other words the summarization of a procedure is independent of the calling context, which is the key to reusing the information. However, there is also a downside to this approach, namely the fact that the analysis has to work on the dataflow transformers and not directly on some dataflow facts (i.e., we compute how the dataflow facts can change). This often makes it more difficult to formulate analyses whose results we can actually compute. The main challenge is that if some domain D satisfies, e.g., the ascending chain condition, when lifted to transformers $D \to D$ it might not satisfy this condition anymore. Fortunately we can still express many analyses. Even for cases like constant propagation where D is usually a mapping from variables to integers/reals, it is possible to define computable variants, i.e., copy- and linear-constant propagation [66, 67]. Obviously whenever D is finite then $D \to D$ will be finite as well. This might seem a bit restrictive, but there are many analyses that satisfy this requirement. In fact, the interprocedural analysis based on graph reachability [64] works on distributive functions $\mathcal{P}(D) \to \mathcal{P}(D)$ where D is required to be some finite set.

Finally, by using flow algebras instead of semirings we achieve the same advantages as presented in Chapter 3, i.e., many analyses that do not form a semiring fit directly in the approach based flow algebras. Recall an example from that chapter (Subsection 3.2.2) about analyses based on bit vector frameworks, where we work with functions $\mathcal{P}(D) \to \mathcal{P}(D)$. A kill-gen analysis where transfer functions are of the form $f_i(l) = (l \setminus k_i) \cup g_i$ can be represented by a pair (k_i, g_i). We additionally have that $\bar{0} = (D, \emptyset)$ and the $\otimes$ operator is defined as:

$$(k_1, g_1) \otimes (k_2, g_2) = (k_1 \cup k_2, (g_1 \setminus k_2) \cup g_2)$$

It is quite easy to see that $\bar{0}$ does not annihilate from both sides, which intuitively

should be unsurprising — the kill-gen analyses are in general not strict.

5.8 Conclusions

Weighted/communicating pushdown systems have been used in many contexts and are a popular approach to interprocedural analysis. However, their requirements with respect to the abstract domain were quite restrictive and did not admit some of the classical analyses directly. In this chapter we have shown that some of the restrictions are not necessary. We have achieved that by reformulating the algorithms for backward and forward reachability. Furthermore, we have proved that they are sound — they always provide a safe over-approximation of the join over all valid paths solution. Provided some additional properties of the abstract domain, we have also shown that those solutions coincide, i.e., the algorithms are complete.

We believe that our results strengthen the connection between the monotone frameworks and the pushdown systems by making it possible to directly express more analyses based on monotone frameworks in the setting of pushdown systems. Moreover, the development does provide some additional flexibility when both designing and implementing analyses using pushdown systems. For instance, the annihilation property might be useful for certain analyses, but now this is the choice of the designer of the analysis and not a strict requirement of the framework. Last but not least, we believe that the chapter improves the understanding of using weighted pushdown systems for interprocedural program analysis.

Chapter 6

Analysis of an aspect-oriented calculus

This chapter presents a novel application of communicating pushdown systems to the analysis of process calculi with *aspect-oriented* features. The problem of analyzing such a process calculi has been encountered in [36, 80] that introduced aspect orientation to coordination languages. It turns out that introducing aspects makes analyzing such languages far more difficult due to the fact that aspects can be recursive — *advice* from an aspect must itself be analyzed by aspects. Therefore, the problem of reachability of various states of a system becomes much more demanding. In this chapter we show how to solve these challenges by using communicating pushdown systems. Even though they have been used mainly for analysis of recursive programs, we show how they can be adapted and used to model aspect-oriented process calculus.

The chapter is organized as follows. First in Section 6.1 we briefly introduce the basic ideas behind aspect-oriented programming and the challenges it introduces in the context of coordination languages. Then in Section 6.2 we introduce the language that will be used throughout the chapter, both for examples and analysis. In Section 6.3 we present how pushdown systems can be used in this new context and demonstrate that they can faithfully model the processes in our language. Furthermore, in Section 6.4 we illustrate our approach on an example. Finally, we conclude in Section 6.5.

Acknowledgement: This chapter is based on [76].

6.1 Introduction

6.1.1 Aspects

Aspect-oriented programming [47] is a successful programming paradigm that is used in many environments and supported by all major programming languages. This includes Java with AspectJ [46] as well as C and C++ [15, 74]. Aspect-oriented programming is often praised for the ability to create modular software and separate cross-cutting concerns. This makes it possible not only to create very modular software but also to add some additional functionality to an existing code base in a non-intrusive fashion. The classical example is of course logging — in order to be useful it should be performed in many different and unrelated parts of the code. Aspects allow to separate the code for logging from the code implementing the program logic. Furthermore, an existing program does not need to be modified in order to add or change the logging mechanism. Recently aspects have been also considered in the context of coordination languages [36, 80], offering similar advantages.

Usually an aspect consists of a pointcut and an advice. A pointcut is basically a pattern that specifies when some join point (e.g., location in a program, some action, etc.) matches the aspect. An advice consists of some additional code or actions that should be executed if the pointcut matches. Often an advice might specify actions that should be performed after, before or instead of the matched one. All of this allows one to easily separate the code for the actual functionality from the code for e.g., logging, which could be specified using aspects.

As already mentioned, aspects have also been used in the context of process calculi — [36] introduced a language called AspectK, which is an extension of the coordination language KLAIM with aspect-oriented features. One of the main motivations behind that work was to use aspects for access control and in doing so separate the access control policy from program logic. However, somewhat surprisingly, the introduction of aspects makes the analysis of such languages much more challenging. This is due to the fact that the advice from an aspect should also be analyzed by all aspects. Therefore, if one allows the advice to contain before and/or after actions, the aspects introduce an additional recursive structure to a program. One of the consequences is that a process can grow arbitrarily large due to advice (we will see an example of this in one of the subsequent sections).

In the following part of this chapter we focus on solving these challenges. For that purpose we define a process calculi allowing concurrent threads to communicate via message-passing. The definition of aspects is quite similar to the one in [36] and allows both before and after actions. Our main contribution is the novel application of communicating pushdown systems in the context of a coordination language with aspect-oriented features. We show that they can be adapted to our context and give us the ability to model arbitrarily large processes as well as to summarize the actions they execute. Furthermore, we present how to go a step further and reason about concurrent systems with aspects. We focus on proving unreachability of certain global states of such systems. The problem of reachability is often of high importance in the context of software verification, because many desired properties of various systems (not only concurrent) can be reduced to the question of reachability of the error states.

6.1.2 Related work

In [80] static analysis techniques have been used to analyze AspectKE (and a programming language AspectKE*), which are based on AspectK. However, these languages do not allow the advice to contain before or after actions, which practically avoids the above mentioned challenges. In this paper we focus on techniques that allow lifting this restriction. In [10, 11] the authors use communicating pushdown systems to analyze concurrent programs with recursive procedures and synchronization-based communication. This approach has been also applied to C programs in [14]. Our approach is based on this work, it does however, have some crucial differences. Clearly the context is quite different as we do not deal with procedural programming languages but process calculi. In particular the source of recursion are the aspects and not the procedures. Consequently we shall use the stack in a completely different way. In [10, 11, 14] the stack is used for storing locations of the programs, whereas we do not even have the concept of location and use the stack to represent the actual process itself (i.e., the action to be executed). In [54] pushdown systems are used to analyze concurrent software, but in a setting of shared memory concurrency. Moreover, this approach is under-approximating with respect to control flow, since it performs the analysis under a context bound. That is, it limits the number of possible context switches that the threads can make.

6.2 Language

Now we will introduce the language that will be used in this chapter. Note that since our focus is on the analysis technique, some of the design decisions have been directed by the ease of presentation. It could be easily extended but that would unnecessarily complicate the analysis. Moreover, we believe that our approach can be adapted to various process calculi with aspect-oriented features.

6.2.1 Syntax and semantics

The language allows for multiple threads running concurrently and communicating via synchronous message passing. The communication is performed in CSP style [39] where the **send** and **receive** actions specify the recipient and sender respectively.

$$
\begin{array}{llcl}
\textit{nets} & N & ::= & N_1 \parallel N_2 \mid c :: P \mid c :: \mathbf{Rec}\mathsf{X}\,.\,P \\
\textit{processes} & P & ::= & \sum_i a_i.Q_i \\
 & Q & ::= & P \mid \mathsf{X} \\
\textit{actions} & a & ::= & \mathbf{receive}(\bar{p})@c \mid \mathbf{send}(\bar{t})@c \mid \\
 & & & \mathbf{if}(e)\ a \mid \mathbf{break} \mid \mathbf{skip} \\
\textit{tests} & e & ::= & t_1 = t_2 \mid t_1 \neq t_2 \mid \mathbf{true} \\
\textit{terms} & t & ::= & c \mid x \\
\textit{patterns} & p & ::= & t \mid !x
\end{array}
$$

In the above rules we use c to denote constants and we will write 0 for a nullary sum. For readability we will write them with an uppercase first letter and variables with a lowercase one.

One, worth mentioning, subtlety about **receive** is that the two actions:

$$\mathbf{receive}(!x)@\mathsf{N} \qquad \text{and} \qquad \mathbf{receive}(x)@\mathsf{N}$$

are quite different — the former will evaluate to itself when ready to execute and will accept any value from the process N and bind it to x. Whereas in the latter case x is an already bound variable and thus the process only accepts this value from N. In other words, if x is bound to C the action will evaluate to $\mathbf{receive}(\mathsf{C})@\mathsf{N}$ and thus only accept C from N.

When talking about the aspects and some other features of our language we will mostly follow [36], which is one of the main motivations for our development.

Thus, before we present the semantics of the language, we will first introduce the well-formedness conditions of the processes. For that purpose we will use functions bv and fv that return the bound and free variables respectively. They are defined in a standard way, e.g.,

$$\mathsf{bv}(\mathbf{receive}(\mathsf{C}, u, !v)@\mathsf{P}) = \{v\}$$

and

$$\mathsf{fv}(\mathbf{receive}(\mathsf{C}, u, !v)@\mathsf{P}) = \{u\}$$

Now consider a pattern $\vec{t} = t_1 \cdots t_k$ that is specified in a **receive** action. We require that:

$$\forall i, j : i \neq j \implies \mathsf{bv}(t_i) \cap \mathsf{bv}(t_j) = \emptyset$$

and

$$\mathsf{bv}(\vec{t}) \cap \mathsf{fv}(\vec{t}) = \emptyset$$

The two rules amount to disallowing bounding the same variable multiple times in one action (first rule) and using the same variable both for pattern matching and bounding new value to it at the same time. In other words both $\mathbf{receive}(!u, !u)@\mathsf{Q}$ and $\mathbf{receive}(!u, u)@\mathsf{Q}$ are not allowed.

The semantics of our language is not surprising and can be expressed in just a few rules.

$$\begin{aligned}
&c :: (\mathbf{send}(t_s)@d + \cdots) \,.\, P \mid\mid d :: (\mathbf{receive}(t_r)@c + \cdots) \,.\, Q \\
&\quad \longrightarrow c :: P \mid\mid d :: Q\theta && \text{if } \mathsf{match}(t_r, t_s) = \theta \\
&c :: (\mathbf{if}(b)\ a + \cdots) \,.\, P \longrightarrow c :: a \,.\, P && \text{if } [\![b]\!] = \mathsf{tt} \\
&c :: (\mathbf{if}(b)\ a + \cdots) \,.\, P \longrightarrow c :: P && \text{if } [\![b]\!] = \mathsf{ff} \\
&c :: (\mathbf{skip} + \cdots) \,.\, P \longrightarrow c :: P \\
&c :: (\mathbf{break} + \cdots) \,.\, P \longrightarrow c :: 0
\end{aligned}$$

We also allow a recursive process, i.e., $\mathbf{Rec}\mathsf{X} \,.\, P$, to execute P, thus we have: $c :: \mathbf{Rec}\mathsf{X} \,.\, P \longrightarrow c ::_P P$. The additional annotation has no influence on the above rules and is carried around so that we can "recreate" the process when the X symbol is reached: $c ::_P \mathsf{X} \longrightarrow c :: \mathbf{Rec}\mathsf{X} \,.\, P$.

In the above rules we have used the function match, which as the name suggests matches the tuples and creates a substitution for the receiving process:

$$\begin{aligned}
\mathsf{match}(\langle\rangle, \langle\rangle) &= id \\
\mathsf{match}(\langle t_1 \cdots t_k\rangle, \langle t'_1 \cdots t'_k\rangle) &= \theta \circ \mathsf{match}(\langle t_2 \cdots t_k\rangle, \langle t'_2 \cdots t'_k\rangle) \\
\text{where } \theta &= \begin{cases} id & \text{if } t_1 \text{ is a constant } (t_1 = c) \text{ and } c = t'_1 \\ [t'_1/x] & \text{if } t_1 \text{ is bounding a variable } (t_1 = !x) \\ \underline{\mathsf{undef}} & \text{otherwise} \end{cases}
\end{aligned}$$

Finally, the nets can evaluate according to the following simple rule:

$$\frac{N_1 \longrightarrow N_1'}{N_1 \mid\mid N_2 \longrightarrow N_1' \mid\mid N_2}$$

and we obviously assume that $\mid\mid$ is associative and commutative.

Note that we also require the nets to be closed, i.e., all variables must be in scope of their defining occurrence.

6.2.2 Aspects

As already mentioned, one of the main features of our language is the presence of aspects. We define them as follows (almost the same as in [36]):

aspects	asp	$::=$	$A[cut;\ e] \triangleq adv$
advice	adv	$::=$	as **break** $\mid$ as **proceed** as
action sequence	as	$::=$	$a \,.\, as \mid \epsilon$
pointcut	cut	$::=$	$c :: a$

We will often call actions before and after **proceed** as before- and after-actions respectively. Informally, the semantics of the aspects says that before executing an action we need to check what aspects apply to it and then combine the advice from them. Checking if an aspect applies to an action amounts to pattern matching against the *cut* and evaluating the applicability condition e associated with the aspect. Note that in case of **receive** action with input, e.g. **receive**$(!x)@N$ we require that the condition does not refer to x. The reason for this is simple — we want to evaluate the condition before the action is executed, so that we can, for instance, disallow it. However, the value of x would be available only after executing the action.

Adding the aspects amounts mostly to following [36] with some minor adjustments to our language. Therefore, we will not repeat that development and only recall the main ideas behind it:

- Every action must be checked by the list of aspects (order matters) and an action might be trapped by multiple aspects.
- If an action matches the pattern of some aspect and the applicability condition evaluates to true, we "inject" the advice into the process.

- The **proceed** in the advice is substituted with the original action, but must be treated carefully and only analyzed by the remaining aspects.
- All actions except for **proceed** and **break** appearing in an advice must also be checked by all the aspects.
- Once an action has been checked by all the aspects (and potentially trapped by some of them) it can be executed.

It is important to emphasize that the checking and trapping by aspects is internal to every process (i.e., not visible from other processes). However, the advice that is offered by aspects can of course change the observable behavior of a process.

The main challenge in allowing the before- and after-actions in advice is that they should be analyzed by all aspects as usual. Otherwise, using aspects for, e.g., access control would not be very useful since the advice would be circumventing the access control policy. The following example shows one of the potential problems that this might pose.

Example 6.1 *Consider the following process with an aspect (for simplicity we skip here the processes* Q *and* Log *and assume that sending anything to them will always succeed).*

$$\mathsf{P} :: \boldsymbol{send}(\mathsf{Test})@\mathsf{Q}$$

$$\mathsf{A}_1[\mathsf{P} :: \boldsymbol{send}(a)@q;\ \boldsymbol{true}] \triangleq \boldsymbol{send}(a)@\mathsf{Log}\,.\,\boldsymbol{proceed}$$

The aspect will trap any ***send*** *action of* P*. However, since the advice will also be analyzed by the aspect, it will trap the* ***send*** *action directed to* Log*. So this example actually demonstrates the possibility of non-termination and a process that can grow infinitely large. Clearly the aspect as defined above is not providing the desired behavior. A better way to specify it is to restrict what actions should be trapped.*

$$\mathsf{A}_1[\mathsf{P} :: \boldsymbol{send}(a)@q;\ q \neq \mathsf{Log}] \triangleq \boldsymbol{send}(a)@\mathsf{Log}\,.\,\boldsymbol{proceed}$$

With this small refinement the aspect will only trap the ***send*** *actions that are not sent to the* Log*. Thus the system will successfully terminate.*

Because our formulation of aspects allows both the before- and after-actions and they need to be analyzed by all aspects themselves, the processes can exhibit a recursive structure. Before we present our solution to this problem, we will introduce the running example of this chapter that will help us illustrate the approach.

6.2.3 Example

We will now present a more involved example that will be used throughout the chapter in demonstrating our approach. Imagine an ATM[1] session — it first receives some credentials from the user and checks the credentials against the information stored on the card. If everything matches it dispenses the cash and informs the bank to deduct the given amount from the account. The following definition models this behavior

$$\begin{aligned}\mathsf{ATM} :: \ & \mathbf{receive}(!\mathit{credentials}, !\mathit{amount})@\mathsf{User}\,.\\ & \mathbf{check}(\mathit{credentials})\,.\\ & \mathbf{send}(\mathit{amount})@\mathsf{User}\,.\\ & \mathbf{send}(\mathit{credentials}, \mathit{amount})@\mathsf{Bank}\end{aligned}$$

where **check** is an internal action that does not involve any communication or synchronization and either executes successfully if the credentials are valid or otherwise terminates the session. This process seems reasonable but we can imagine that in order to increase the security of this solution one could add aspects that actually confirmed the credentials with the bank.

$$\begin{aligned}\mathsf{A}_1[\mathsf{ATM} :: \mathbf{check}(c);\ \mathbf{true}] &\triangleq \mathbf{proceed}\,.\,\mathbf{send}(c)@\mathsf{Bank}\,.\\ &\quad\ \mathbf{receive}(!a)@\mathsf{Bank}\\ \mathsf{A}_2[\mathsf{ATM} :: \mathbf{receive}(!a)@\mathsf{Bank};\ \mathbf{true}] &\triangleq \mathbf{proceed}\,.\\ &\quad\ \mathbf{if}(a = \mathsf{Abort})\ \mathbf{send}(\mathsf{ErrorMessage})@\mathsf{User}\,.\\ &\quad\ \mathbf{if}(a = \mathsf{Abort})\ \mathbf{break}\end{aligned}$$

The above two aspects make the additional check of credentials with the bank (after checking locally) to improve the security. Obviously we want the ATM session to terminate (with an error message) when this check fails.

Apart from that we need to define the process modelling the Bank

$$\begin{aligned}\mathsf{Bank} :: \ & (\mathbf{receive}(!\mathit{credentials})@\mathsf{ATM}\,.\,\mathbf{send}(\mathsf{Ok})@\mathsf{ATM}\,.\\ & \quad \mathbf{receive}(\mathit{credentials}, !\mathit{amount})@\mathsf{ATM})\\ & +(\mathbf{receive}(!\mathit{credentials})@\mathsf{ATM}\,.\,\mathbf{send}(\mathsf{Abort})@\mathsf{ATM})\end{aligned}$$

We do not define the process for User since it does not actually bring anything interesting to the example.

Now having such a system, one of the things that we would like to guarantee is that whenever the bank aborts a transaction, the ATM will not dispense the

[1] Automated Teller Machine

cash. In other words we want to ensure that both the bank and the ATM have a consistent view on the transaction. To achieve that we will use communicating pushdown systems.

6.3 Pushdown systems

6.3.1 Modelling processes with aspects

6.3.1.1 Basic idea

Pushdown systems are commonly used in the context of recursive programs. They provide a very natural way to model such systems since the unbounded stack can be used to store the return addresses of the called procedures. However, in our scenario there are neither procedures nor any notions of addresses or program locations. Nevertheless, as we have already mentioned, the main difficulty of analysis of our process calculus lies in the recursive structure that can be introduced by aspects. And in some ways the behavior of aspects does resemble that of a stack. Consider, for instance, a process $A \,.\, B$, if the first action A is trapped by an aspect that gives advice $C \,.\, D \,.\, \mathbf{proceed}$ then we suddenly have a process $C \,.\, D \,.\, A \,.\, B$ (where A should not be considered by this aspect again). But this can be thought of as a stack — we push two additional actions on it and then want to continue the execution starting from the top. So after executing C we are left with a process that looks like $D \,.\, A \,.\, B$ — this clearly corresponds to popping C from the stack. Therefore, the main twist of our approach is to use the stack to characterize the process itself (i.e., the actions that are to be executed).

There are a number of considerations that need to be taken into account to faithfully model the behavior introduced by aspects. One of the problems is ensuring that an aspect will not trap the same action multiple times. As presented in the previous section, the **proceed** represents the trapped action but it should only be checked by aspects after the current one. Our solution to this problem is to embed some additional information in the stack to indicate what aspect should consider the given action next. Therefore, each action will also contain the information about the aspects that can potentially trap it next.

Since we already use the stack for storing the process along with the information about aspects, we can often improve the precision of our analyses by embedding in the stack some information about the communication that takes place. The most natural choices are the sender and receiver as well as the contents of the

tuples that are sent or received. However, at this point we encounter here a limitation of the original formulation of CPDS — they were defined to handle synchronization actions. This is a bit problematic since we want to work with message passing and the contents of messages that are being sent is essential for determining whether certain configurations are reachable. This is simply due to the fact that the control flow of the processes is often based on the received values. Our current solution is to instantiate all the tuples and generate rules for all possibilities. We usually do not want to consider all possible constants and all possible tuples that can arise at runtime, since often just a subset of them along with some abstraction of the remaining ones can be enough to prove the property of interest. We will call them abstract constants and abstract tuples — since they abstract away from some of the possible runtime values. For instance, in our example of an ATM and bank we are probably interested in whether Ok and Abort are sent and received. On the other hand, information such as the amount of money to be withdrawn, is not that important due to the fact that it does not influence the control flow of the processes.

Let us first denote the set of all constants as Const and the set of abstract constants as $\widehat{\mathsf{Const}}$. Similarly we will denote the set of all possible tuples as $\widehat{\mathsf{Tuples}}$ (note that this is also a finite set — any program will have a maximum arity of tuples that it sends/receives). With all of the above considerations, we define the stack alphabet Γ in the following way:

$$\Gamma = ((\mathsf{Proc} \times \mathsf{Proc} \times \widehat{\mathsf{Tuples}}) \cup \mathsf{Internal}) \times (\mathsf{Asp} \cup \{\checkmark\})$$

where Asp is the set of aspects and $\checkmark$ is a special symbol indicating that all aspects have already analyzed the given action, whereas the first component represents either a communication action or an internal one — Proc is the set of processes, $\widehat{\mathsf{Tuples}}$ is the set of all possible abstract tuples that are sent over the channels in the given system and $\mathsf{Internal}$ are internal actions of the process. This last set will usually only include the **break** and **skip** actions and exclude the **if** one — we will represent the possibility of executing an action implicitly using the pushdown rules. The definition of pushdown systems requires that all these sets are finite but obviously concrete processes are always finite and so will be the pushdown system we generate.

Furthermore, we need to define the set of abstract actions. In our case this is actually quite similar to the stack elements. We define them as follows:

$$\widehat{\mathsf{Act}} = (\mathsf{Proc} \times \mathsf{Proc} \times \widehat{\mathsf{Tuples}}) \cup \{\tau\}$$

The intuition here is that we do want to know the sender (first component of the tuple), the receiver (the second component) and what is communicated (the third component). This information will be essential in the subsequent

section on communicating pushdown systems. Apart from that we also need to accommodate internal actions that are not important from the point of view of synchronization with other processes — thus the inclusion of τ that has the property that $a\tau = \tau a = a$.

Notice that in some cases we do not actually know what is sent/received in a given action (e.g. in **receive**$(!x)@\mathsf{N}$ we do not know what x might be). In such cases we can simply generate rules for all the possibilities. However, there are many ways of improving this, for instance, it should be possible to generate such possibilities lazily, i.e., if some action is never pushed on a stack, we do not really need to add rules to pop it. Another example would be if we can determine that some constant is sent only between two processes, then we do not have to consider it when generating rules for actions of other processes.

Now let us get back to our example. Since we are only interested in Ok and Abort constants and the maximum arity of a tuple send by any process is equal to two, our set of abstract tuples will be:

$$\widehat{\mathsf{Tuples}} = \{(c) \mid c \in \mathsf{C}\} \cup \{(c_1, c_2) \mid c_1 \in \mathsf{C},\ c_2 \in \mathsf{C}\}$$

where $\mathsf{C} = \{\mathsf{Ok}, \mathsf{Abort}, *\}$ and $*$ stands for any constant other than Ok or Abort.

6.3.1.2 Rules for creating processes.

To create a process the first thing that we do is to push all its actions on the stack. So if we have a process $\mathsf{P} :: a_1 \,.\, a_2 \,.\, a_3$ then we create a rule

$$\langle \mathsf{P},\ \Box \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{P},\ a_1\ a_2\ a_3 \rangle$$

where $\Box$ is a "start" symbol that can be used for creating the initial set of configurations (we will explain that later on). We will use this symbol for, e.g., the *Post** query (such as: what are all the successors of $\langle \mathsf{P}, \Box \rangle$). Furthermore, this gives us a very nice way to handle the recursion of the processes. We can use X symbol as the start symbol and whenever X is at the top of the stack, the recursion will be handled automatically. Consider the following process: $\mathsf{Q} :: \mathbf{Rec}\mathsf{X} \,.\, a_1 \,.\, a_2 \,.\, \mathsf{X}$ and the initial rule that we create:

$$\langle \mathsf{Q},\ \mathsf{X} \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{Q},\ a_1\ a_2\ \mathsf{X} \rangle$$

The moment a_1 and a_2 are popped from the stack (i.e., executed) the initial rule will apply again and the process will be "recreated".

Apart from that we need to be able to handle choice. We do not have any abstract actions that would express choice since the control flow is handled by

the pushdown rules themselves. Therefore, we create all possible linear shapes of the process. For instance when generating the initial rules for process $\mathsf{P} :: a_1 \,.\, (a_2 + a_3)$ we would create:

$$\langle \mathsf{P},\ \square \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{P},\ a_1\ a_2 \rangle \qquad \langle \mathsf{P},\ \square \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{P},\ a_1\ a_3 \rangle$$

For the ATM in our example we can generate a set of rules for all choices of $x \in \mathsf{C}$ and $y \in \mathsf{C}$:

$$\left\{ \begin{array}{lll|l} \langle \mathsf{ATM},\ \square \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{ATM}, & (\mathsf{User}, \mathsf{ATM}, (x, y), \checkmark) & & \\ & (\mathbf{check}(x), \mathsf{A}_1) & & x \in \mathsf{C} \\ & (\mathsf{ATM}, \mathsf{User}, (y), \checkmark) & & y \in \mathsf{C} \\ & (\mathsf{ATM}, \mathsf{Bank}, (x, y), \checkmark)) \rangle & & \end{array} \right\}$$

As already mentioned, we can often be much smarter about generating the rules and create only a subset of the above (for instance Abort is never sent between User and Bank).

6.3.1.3 Rules for aspects

When generating the rules for the aspects we often have sufficient information in the stack element of the pointcut to be able to decide whether the aspect traps it or not. In the example above we could easily tell that some of the actions could never be trapped by any of our aspects (the aspects in the example trap only actions of the ATM). However, if we do not know whether the aspect will trap the action, we simply over-approximate and generate rules for both possibilities. One of the essential parts of generating the rules is to update the component of the stack that tracks what aspect should analyze the action next. In other words, if an action a is trapped by aspect A_1 then the **proceed** of the aspect should be the same action a but annotated with the next aspect. To make that clear, let us consider the internal **check** action of our ATM:

$$\left\{ \begin{array}{lll|l} \langle \mathsf{ATM},\ (\mathbf{check}, (x), \mathsf{A}_1) \rangle \overset{\tau}{\hookrightarrow} \langle \mathsf{ATM}, & (\mathbf{check}, (x), \checkmark) & & \\ & (\mathsf{ATM}, \mathsf{Bank}, (x), \checkmark) & & x \in \mathsf{C} \\ & (\mathsf{Bank}, \mathsf{ATM}, (y), \mathsf{A}_2) \rangle & & y \in \mathsf{C} \end{array} \right\}$$

As can be seen above, we have the internal action **check** with aspect A_1 on the left-hand side of the pushdown rule, but on the right-hand side we annotate it with $\checkmark$ as there are no more aspects that can match. This ensures that **check** will not be trapped by this aspect again.

Moreover, we must also handle the **if** conditions. Since we generate rules for various combinations of constants, we can often determine whether a condition is true at the stage of generating the rules. And if so, we can generate rules just for the right branch. However, in general this is not always possible. In such situations we do the same as in the case of aspects, i.e., generate the rules for both cases — one if the condition is true and one if it is false. This again corresponds to over-approximating the control flow. Therefore, from the point of view of precision, it might be beneficial to include in the set of abstract constants the ones that are used for comparisons.

6.3.1.4 Rules for executing actions

All of the above rules do not model the execution of any actions (and thus are considered as internal actions and labeled with τ). Execution in our context is nothing else than simply popping a stack element. So in general we simply create rules of the form $\langle p, a\rangle \overset{l}{\hookrightarrow} \langle p, \epsilon\rangle$ for all possible actions a that are annotated with $\checkmark$, where l is equal to either τ if a is an internal action or a otherwise. An example from ATM is as follows:

$$\left\{ \langle \mathsf{ATM},\ (\mathsf{User}, \mathsf{ATM}, (x, y), \checkmark)\rangle \xhookrightarrow{(\mathsf{User},\mathsf{ATM},(x,y))} \langle \mathsf{ATM},\ \epsilon\rangle \;\middle|\; x \in \mathsf{C}\ y \in \mathsf{C} \right\}$$

which corresponds to execution of all actions where ATM receives a two-tuple from the user. Note that we require that the actions are annotated with $\checkmark$, which ensures that the action cannot be executed before all aspects have been considered.

6.3.1.5 What are $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$?

Once we have all the pushdown rules, we can ask for all successor/predecessor configurations. However, the notion of a configuration in our setting is slightly different than usual — it is actually the process itself (the stack contains all the actions to be executed). Therefore, the $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ automata will correspond to all possible shapes of the process itself. Let us consider an example: we want to compute all the possible "futures" of a process. So the initial automaton $\mathcal{A}$ should represent just a singleton language, in case of ATM we would have just one transition $\mathsf{ATM} \xrightarrow{\square} q_f$ (where ATM is an initial state and q_f is a final one). Now running the *Post** algorithm on it will use the rule to construct the process and then execute the actions and "calling" the aspects. The resulting $\mathcal{A}_{post^*}$ automaton will represent all those configurations, so it will describe what the process can look like in the future (i.e., grow due to advice, shrink

due to executing the actions, etc.). The weights of the transitions of $\mathcal{A}_{post^*}$ will capture the abstraction of the communication the process has performed to get to the given configuration. This can be used to, e.g., ask for the summarization of the communication of the process when there is nothing on the stack (i.e., corresponding to the termination of the process).

6.3.2 Communicating pushdown systems

For now we have considered only a single pushdown system, which corresponds to a single process. Since we are interested in the behavior of the whole concurrent system, we will use the communicating pushdown systems. However, CPDS in the original formulation do not differentiate between send and receive actions and focus on synchronization actions only (no message passing). Therefore, we slightly change the definition of the transition system created by a CPDS. A global configuration of CPDS is, as before, a tuple $g = (c_1, \ldots, c_n)$ of configurations of $\mathcal{P}_1, \ldots, \mathcal{P}_n$. But the relation $\overset{a}{\Longrightarrow}$ is generalized to global configurations in a slightly different way:

- $g \overset{\tau}{\Longrightarrow} g'$ if there is $1 \leq i \leq n$ such that $c_i \overset{\tau}{\Longrightarrow} c'_i$ and $c'_j = c_j$ for all $j \neq i$
- $g \overset{(s,r,t)}{\Longrightarrow} g'$ if there are $i \neq j$ such that $c_i \overset{(s,r,t)}{\Longrightarrow} c'_i$ and $c_j \overset{(s,r,t)}{\Longrightarrow} c'_j$ ("s" stands for sender, "r" for receiver and "t" for tuple). Finally, for all $k \neq i \wedge k \neq j$ we have that $c'_k = c_k$.

This ensures that the processes agree on the direction of the communication. Also note that in our formulation the processes specify the sender or receiver, which corresponds to requiring that there is a unique uni-directional channel between every two processes. But that could be generalized to allow arbitrary channels and, as noted in [10, 11], one would have to transform the program into one where every channel is used by exactly two processes (this is due to the shuffle operator).

Of course, since the languages generated by each of the processes can be context-free, we still want to perform some abstraction. In our situation we have decided to use the ith-prefix abstraction from [14] that, as the name suggests, considers the prefixes of communication traces.

6.4 Analysis

6.4.1 The non-recursive example

We have implemented this abstraction and used the WALi (Weighted Automata Library) [48] for computing the $Post^*$ weighted automaton. Currently our implementation can be seen as a small library on top of WALi that offers higher level API capable of generating the required pushdown rules. This includes generating all possible tuples of for a given set of abstract constants as well as reducing the push rules using more than two stack elements on their right hand side.

Let us consider the analysis for our example of Bank and ATM. With that in mind we use $\widehat{\mathsf{Const}} = \{\mathsf{Ok}, \mathsf{Abort}, \mathtt{*}\}$ and set $i = 3$.

Since the resulting graphs are simply too large to include here, we have simplified the rules by avoiding generating the rules for all possible abstract constants (i.e., we assume that the user always sends `*`, therefore this does not compromise the results). The results of $Post^*$ algorithm are presented in Figure 6.1 and Figure 6.2.

The annotations on the edges are pairs of stack element (first component; `*` denotes an ϵ transition) and the weight of the transition (second component). Moreover, `EA` stands for ✓ and `LabelAbort()` is an additional internal action that we inserted just after the action sending Abort to make it easier to see in the summarization of bank's actions at this point, which is:

$$\{\ \texttt{ATMBank(*)BankATM(Abort)}\ \}$$

Note that for clarity of presentation we have not used the shuffle operator on the above set.[2] In general we should include there all words (with up to three actions) created from the above and interspersed with the possible communication actions between the ATM and the User.

Turning to the ATM, we can see that the process successfully dispenses the money and is about to inform the bank about the withdrawal with the following summarization of its communication:

$$\{\ \texttt{UserATM(**)ATMBank(*)BankATM(Ok)}\ \}$$

Now it should be clear that the intersection of the communication of the bank when it sends the abort message and the ATM when it dispenses the cash is

[2] Recall that in [10, 11] it is used to account for communication that does not involve the given process.

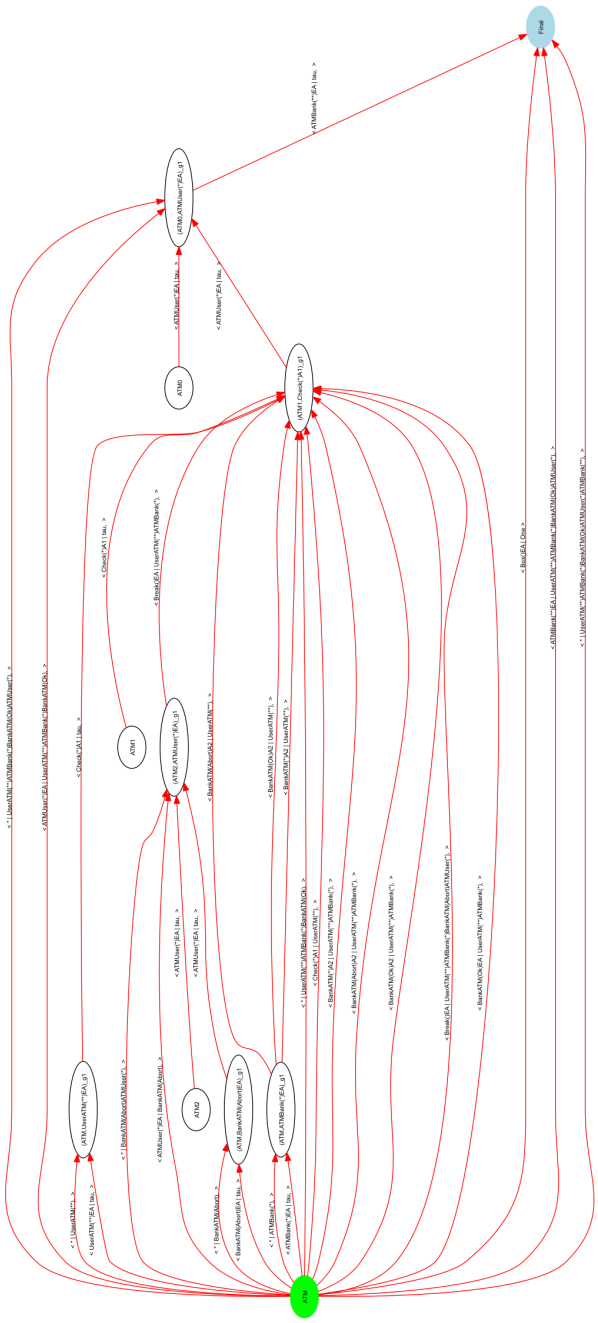


Figure 6.1: Result of a simplified analysis of the ATM.

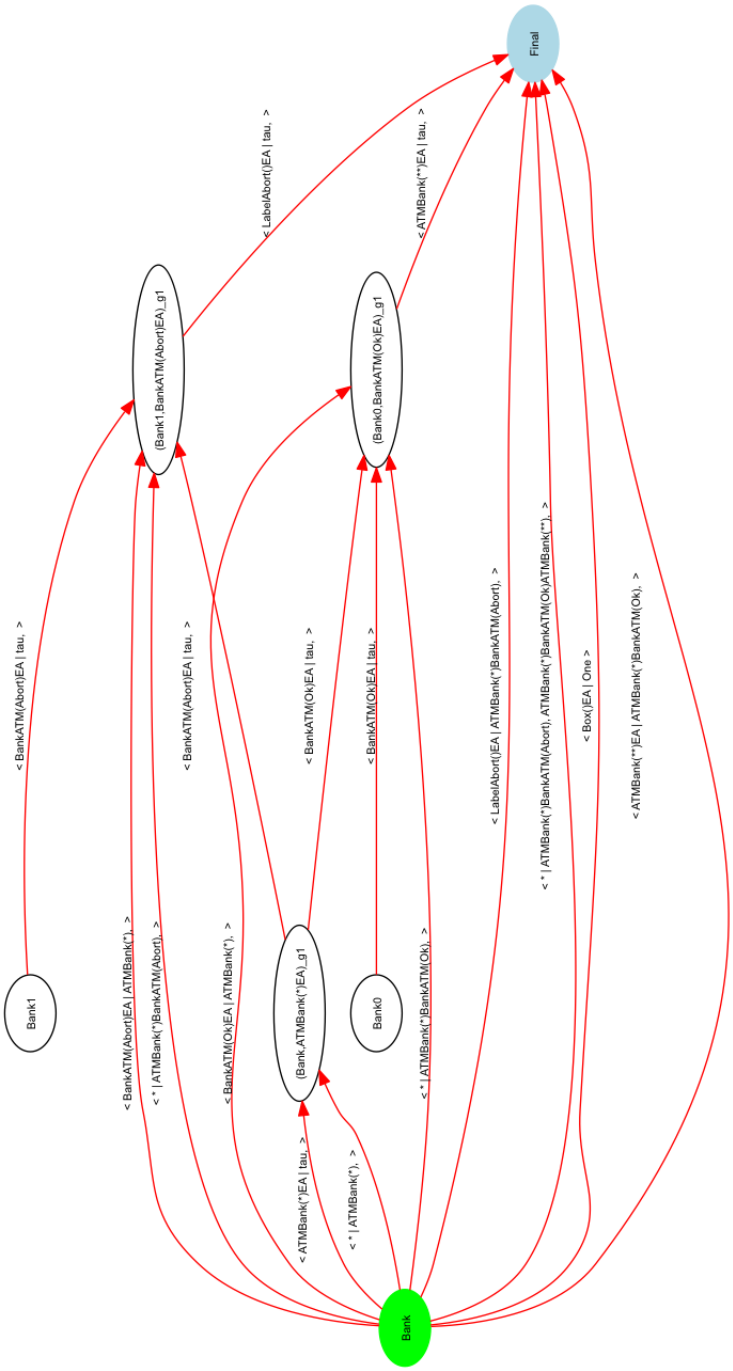


Figure 6.2: Result of a simplified analysis of the bank.

empty — `BankATM(Ok)` and `BankATM(Abort)` do not match. This means that it is impossible for the bank and the ATM to reach this error configuration. In other words the cash will never be dispensed in a situation where the bank aborts the transaction.

6.4.2 The recursive example

Now let us consider the same example, but modified such that both the bank and the ATM are recursive processes. In other words we want to model not just one session, but many subsequent ones. The modifications are quite simple and amount to just adding the recursive annotations in the right places. The updated ATM is:

$$
\begin{aligned}
\mathsf{ATM} :: \ & \mathbf{Rec}\mathsf{X}\,. \\
& \mathbf{receive}(!\mathit{credentials}, !\mathit{amount})@\mathsf{User}\,. \\
& \mathbf{check}(\mathit{credentials})\,. \\
& \mathbf{send}(\mathit{amount})@\mathsf{User}\,. \\
& \mathbf{send}(\mathit{credentials}, \mathit{amount})@\mathsf{Bank}\,.\,\mathsf{X}
\end{aligned}
$$

The aspects do not need to be modified and are repeated below for the reader's convenience:

$$
\begin{aligned}
\mathsf{A}_1[\mathsf{ATM} :: \mathbf{check}(c);\ \mathbf{true}] &\triangleq \mathbf{proceed}\,.\,\mathbf{send}(c)@\mathsf{Bank}\,. \\
&\quad\ \mathbf{receive}(!a)@\mathsf{Bank} \\
\mathsf{A}_2[\mathsf{ATM} :: \mathbf{receive}(!a)@\mathsf{Bank};\ \mathbf{true}] &\triangleq \mathbf{proceed}\,. \\
&\quad\ \mathbf{if}(a = \mathsf{Abort})\ \mathbf{send}(\mathsf{ErrorMessage})@\mathsf{User}\,. \\
&\quad\ \mathbf{if}(a = \mathsf{Abort})\ \mathbf{break}
\end{aligned}
$$

Finally, the bank should restart communication once it has handled an ATM request.

$$
\begin{aligned}
\mathsf{Bank} :: \ & \mathbf{Rec}\mathsf{X}\,. \\
& (\mathbf{receive}(!\mathit{credentials})@\mathsf{ATM}\,. \\
& \ \ \mathbf{send}(\mathsf{Ok})@\mathsf{ATM}\,. \\
& \ \ \mathbf{receive}(\mathit{credentials}, !\mathit{amount})@\mathsf{ATM}\,.\,\mathsf{X} \\
& +\mathbf{receive}(!\mathit{credentials})@\mathsf{ATM}\,. \\
& \ \ \mathbf{send}(\mathsf{Abort})@\mathsf{ATM}\,.\,\mathsf{X})
\end{aligned}
$$

The analysis is similar to the one before, except for the fact that we generate some additional rules to handle the recursion. However, the ith-prefix abstraction is not capable of verifying the example due to the fact that there will be traces of previous successful sessions that will be i communication actions long, and thus the last session (the one that we are asking about) will have the Abort message truncated. Fortunately, the ith-suffix abstraction works due to the same reasons why i^{th}-prefix fails — in this case the most important part of the trace is that of the last Abort which is never truncated. The results of the analysis are depicted on Figure 6.3 and Figure 6.4. We can read the weights of the transitions, but this time we will have to look at two transitions (the X symbol, on the graphs denoted as `Box`, is always at the bottom of the stack). The results for the ATM:

$$\{\tau, \texttt{BankATM(Ok) ATMUser(*) ATMBank(**)}\}$$
$$\otimes$$
$$\{\texttt{ATMBank(*) BankATM(Ok) ATMUser(*)}\}$$
$$=$$
$$\{\texttt{ATMBank(*) BankATM(Ok) ATMUser(*)}\}$$

And the results for the bank:

$$\left\{\begin{array}{c}\tau,\ \texttt{ATMBank(*)BankATM(Abort)},\\ \texttt{ATMBank(*)BankATM(Ok)ATMBank(**)},\\ \texttt{ATMBank(**)ATMBank(*)BankATM(Abort)},\\ \texttt{BankATM(Abort)ATMBank(*)BankATM(Abort)}\end{array}\right\}$$
$$\otimes$$
$$\left\{\ \texttt{ATMBank(*)BankATM(Abort)}\ \right\}$$
$$=$$
$$\left\{\begin{array}{c}\texttt{ATMBank(*)BankATM(Abort)},\\ \texttt{BankATM(Abort)ATMBank(*)BankATM(Abort)},\\ \texttt{ATMBank(**)ATMBank(*)BankATM(Abort)},\\ \texttt{BankATM(*)ATMBank(*)BankATM(Abort)}\end{array}\right\}$$

Note that, in general, we should use the shuffle operator that in the case of bank would account for the communication between the ATM and the user. However, we skip that for the clarity of the presentation.

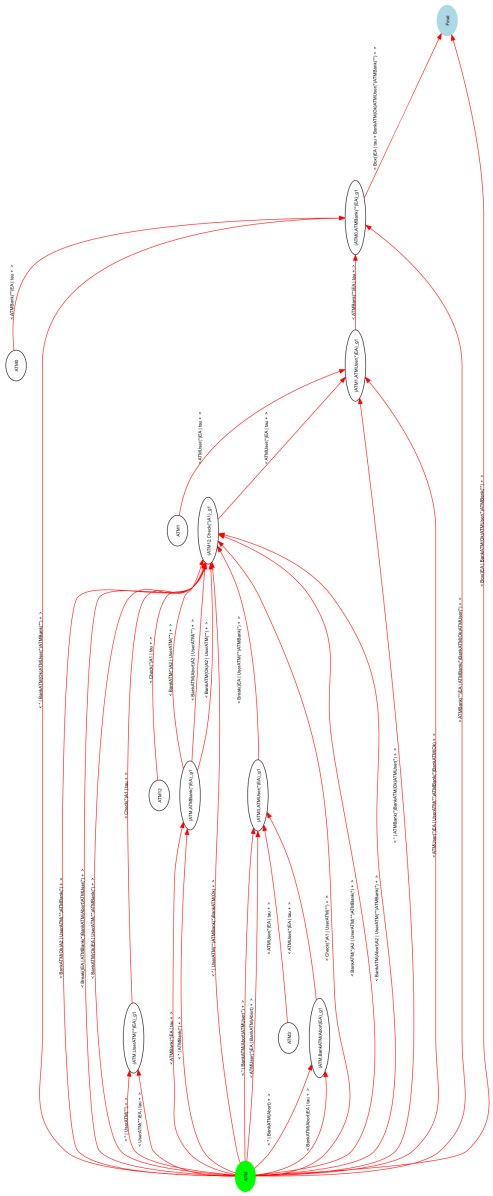


Figure 6.3: Result of a simplified analysis of recursive ATM.

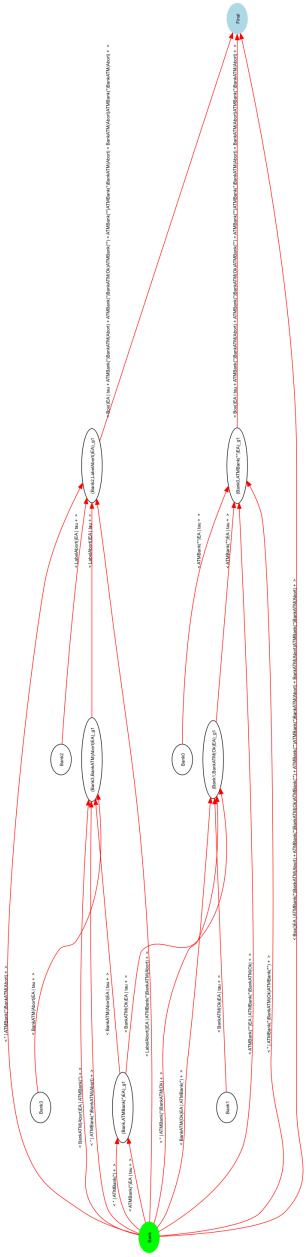


Figure 6.4: Result of a simplified analysis of recursive Bank.

6.4.3 Discussion

There are a few interesting details of our analysis that we will review in this section. First of all, note that when generating the rules for creating the process or "calling" an aspect we introduce some fresh states in a way that allows us to maintain the relationship of the variables used in different actions (i.e., if we receive a value and bind it to x and then send x, we want to maintain that the same value is first received and then sent). Consider the following example:

$$\mathsf{P} :: \mathbf{receive}(!x)@\mathsf{Q}\,.\,\mathbf{send}(\mathsf{B})@\mathsf{Q}\,.\,\mathbf{send}(x)@\mathsf{Q}$$

and assume that $\widehat{\mathsf{Const}} = \{\mathsf{A}, \mathsf{B}\}$. Since we want to push more than two elements on the stack, we will introduce some fresh states and generate the following rules:[3]

$$\begin{aligned}
\langle P, \square\rangle &\overset{\tau}{\hookrightarrow} \langle P_1, (\mathsf{P}, \mathsf{Q}, (\mathsf{B}))(\mathsf{P}, \mathsf{Q}, (\mathsf{A}))\rangle \\
\langle P_1, (\mathsf{P}, \mathsf{Q}, (\mathsf{B}))\rangle &\overset{\tau}{\hookrightarrow} \langle P, (\mathsf{Q}, \mathsf{P}, (\mathsf{A}))(\mathsf{P}, \mathsf{Q}, (\mathsf{B}))\rangle \\
\langle P, \square\rangle &\overset{\tau}{\hookrightarrow} \langle P_2, (\mathsf{P}, \mathsf{Q}, (\mathsf{B}))(\mathsf{P}, \mathsf{Q}, (\mathsf{B}))\rangle \\
\langle P_2, (\mathsf{P}, \mathsf{Q}, (\mathsf{B}))\rangle &\overset{\tau}{\hookrightarrow} \langle P, (\mathsf{Q}, \mathsf{P}, (\mathsf{B}))(\mathsf{P}, \mathsf{Q}, (\mathsf{B}))\rangle
\end{aligned}$$

It is important to note that we introduce two different states P_1 and P_2 in order to capture that whatever value the process receives in the first action, it sends it in the last one (i.e., the value of x after the first action is the same as its value in the last one). However, it is possible to add fewer new states at the expense of precision, i.e., we could simply ignore the relation between the rules and merge the P_1 and P_2 states. The resulting system would allow for traces like receiving A and then sending B twice, which is clearly a behavior not exhibited by the original system. We believe that in many situations this lack of precision might be too significant, nevertheless, it does have a cost as the complexity of the Pre^* and $Post^*$ algorithms depends on the number of control locations as well as the number of pushdown rules. Thus, the ability to decrease it (at least in certain parts of the programs) gives some additional flexibility in modelling the processes.

Apart from that, it is important to emphasize that, even though we have demonstrated the approach on the ith-prefix/suffix abstractions, it works for any abstractions allowed by the CPDS. This includes, e.g., all the abstractions defined

[3] This only shows the rules when the process receives A, we do not show the ones for B since they would be almost the same.

in [10, 11]. Furthermore, we have only focused here on the communicating pushdown systems, but our approach of modelling the processes is applicable to any pushdown systems, including the weighted pushdown systems (WPDS) [66, 67]. The exact choice depends on the details of the process calculus in question as well as the properties one wants to verify.

Finally, we should also mention that there are a number of remaining challenges of our approach with scalability being one of the main concerns, especially when large sets of abstract constants are involved. This is not really a problem with our adaptation of CPDS for aspect-oriented calculus, but more with the design of CPDS. Since the Pre^* or $Post^*$ analysis of each thread is performed independently, there is no information of what can be sent between the processes. Our solution to that problem is to generate all the possibilities (using the set of abstract constants) thus reducing the problem to the one considered in [10, 11]. It should be clear that we have a combinatorial explosion of the number of rules with respect to the arity of the tuples. Of course, some preliminary analysis could make it possible for us to avoid generating some of the rules. To make matters worse, a larger set of abstract constants also increases the space complexity of abstract domains involved, thus making the $\otimes$ and $\oplus$ operations more expensive. Solving it is one of our motivations behind the work described in Chapter 7, where we present a symbolic abstract domain that is capable of capturing the constraints imposed on the values being communicated.

6.5 Conclusions

In this chapter we have considered a process calculus with aspect-oriented features, which is equipped with message-passing primitives. We believe that aspect-orientation has a lot to offer not only in the area of programming languages but also in the context of coordination languages. In particular it gives us the ability to create very modular systems and separate unrelated functionality, which should make it easier to model complex systems.

However, the addition of aspects with advice allowing before and/or after actions leads to some interesting challenges. Those additional actions make it possible for a process to "grow" — one action trapped by an aspect can result in advice consisting of two or more actions. And since the advice itself is analyzed by aspects, the processes can exhibit a recursive structure and become arbitrarily large. Obviously this makes it much more difficult to analyze such systems. Our main contribution is to present an approach that is capable of solving analysis problems is such a context. To achieve this we used a technique from software model checking, namely communicating pushdown systems. Even though it is

used mainly for analysis of recursive programs, we managed to adapt it to our setting. It proved to be a very useful and quite a flexible tool, able to provide us with descriptions of processes that can be arbitrarily large. Moreover, with the right abstraction, we can compute the summarization of its communication actions, allowing us to reason about the reachability in systems of concurrent threads. Since many safety problems can be reduced to reachability of error states, our approach can be used for verification purposes of such systems.

We believe that this approach can be adapted to various process calculi that use aspect-oriented paradigm. Moreover, our approach to modelling of such processes using pushdown systems makes it possible to build analyses using both communicating and weighted pushdown systems.

There are also some interesting future challenges. For instance, the question of how far can we extend the language and still be able to model it using pushdown systems. Moreover, from the point of view of efficiency we would prefer to generate only a small number of rules. On the other hand, to achieve better precision we would like to include as much information in the rules as possible. There is clearly a lot of room for experiments with various approaches and trade-offs depending on the situation.

Finally, in this chapter we have encountered an interesting problem with the use of the communicating pushdown systems for the analysis of message passing. It arises due to the fact that the control flow often depends on the actual values being sent between processes and achieving good precision while using abstract domains such as i^{th}-prefix/suffix might require generating a much larger number of pushdown rules. We will explore this in the Chapter 7 where we will try to capture the constraints imposed by the control flow in the abstract domain itself.

Chapter 7

Symbolic prefix/suffix abstractions

In this chapter, we focus on some of the challenges mentioned in Chapter 6. We have noted that i^{th}-prefix/suffix abstractions [14] do not fit very well in the context of message passing (to achieve good precision one often has to generate many more pushdown rules). In this chapter we present an abstract domain based on the ith-prefix/suffix abstractions that symbolically encodes the contents of the message as well as constraints that refer to them. Furthermore, it forms a semiring and thus can be used with, e.g., communicating pushdown systems. We develop a data structure optimized for low memory usage, as well as efficient algorithms for the $\oplus$ and $\otimes$ semiring operations. To handle the constraints and feasibility of communication traces, it leverages modern SMT solvers. Furthermore, it allows to easily adjust the amount of information considered and thus offers flexibility with respect to the performance-precision compromise. Finally, our implementation can be easily used with the Weighted Automata Library (WALi) [48].

The structure of the chapter is as follows. We first motivate the development in Section 7.1. Then we discuss the new domain in Section 7.2. In Section 7.3 we turn to the design of the data structure. We discuss the slightly more challenging analysis of reachability in Section 7.4. Finally, we present some experiments using the domain in Section 7.5 and conclude in Section 7.6.

7.1 Introduction

7.1.1 Motivation

Concurrency is increasingly important not only in programming but also in the modelling of various systems. However, it is also known to be extremely difficult, especially in the context of shared-memory. Therefore, in many situations concurrency based on message passing is preferred and advertised as more maintainable, e.g. in Erlang [24], Go [34] or Haskell [37]. Moreover, it is often used in various process calculi that are commonly used for modeling purposes. Even though it is commonly considered easier for programmers to understand, it may prove harder to perform analysis and verification of systems based on message passing. Especially when the processes have some recursive structure, which ideally should be handled by performing context-sensitive analysis. Unfortunately, as shown in [63], an exact analysis that is both context-sensitive and synchronization-sensitive is undecidable. In [10, 11] a generic approach to the analysis of concurrent and recursive processes with synchronization was proposed (we have discussed it in Chapter 4). The problem of undecidability was sidestepped by employing various abstractions of the communication between processes. Later the approach has been successfully used in [14] to analyze C programs and introduced for that purpose the ith-prefix abstraction, which considers the prefixes of synchronization actions. However, neither of the papers consider the problem of message passing where the contents of communication might be essential to verify the reachability of certain states of the system. In the case when the domain of messages is finite, one can reduce the problem of message passing to synchronization by considering all possible values that can be sent. But this naïve approach leads to a blowup that might be quite severe when the domain has many elements. Therefore, we propose a more sophisticated way of solving this problem using symbolic representations.

7.1.2 Contributions

We introduce a new abstract domain building on the already mentioned i^{th}-prefix/suffix abstractions. The new domain symbolically represents the messages that can be sent between the processes. Moreover, it includes various constraints that can be imposed on them by the processes due to the control flow or pattern matching. Since our abstract domain is parametrized by a bounding parameter i, it allows to easily adjust the precision of the analysis for the given problem or even do that dynamically — start with low values of i and try to increase them if needed. By combining it with the ideas presented in [10, 11] we

have the ability to deal with recursive processes, pose queries about the analysis results for various stack contents and then prove unreachability of error states in systems of concurrent processes. We also consider the problem of efficient implementation of our abstract domain, including the representation of the language and the basic operations on it. This is essential since we often have to represent an exponential number of traces. Finally, due to our extensions we need to create a new and somewhat more complex, procedure for proving unreachability in our concurrent system. This must obviously include a way to handle the constraints that are imposed on the contents of messages. We have decided to take advantage of modern SAT and SMT solvers, which nicely fit into our approach and provide the flexibility to deal with various kinds of constraints and messages.

7.1.3 Related work

The papers [10, 11] introduced the approach of computing abstractions of communication of recursive processes and then checking their "intersection" to prove unreachability of error states. The approach has been subsequently used in [14], where the authors proposed their own abstract domains: i^{th}-prefix and i^{th}-suffix abstractions. Both of them, combined with CEGAR (Counter-Example Guided Abstraction Refinement), turned out to be practical for analysis of C programs and allowed the authors to discover a previously unknown bug in a driver for the Windows operating system. In [76] we have used the ith-prefix/suffix abstractions for analysis of process calculi equipped with aspect-oriented features, where we encountered the problem of insufficient information about the contents of the communication. Apart from that, in recent years a lot of attention has also been directed towards methods that perform some under-approximation during the analysis. The basic idea is to avoid the issue of undecidability by bounding, e.g., the number of context-switches [61, 55], where the authors focused on the problem of shared memory concurrency and not message passing. A different approach bounds the number of phases in the communication [8], which allows to transform a concurrent system into a sequential one. Another recent work imposes the requirement that the communication network is acyclic [2]. Both of these approaches are based on under-approximation, consider recursive programs with asynchronous message passing, and require that the domain of messages is finite. We are also considering recursive programs, but our approach performs only over-approximation, does not impose the restriction on the finiteness of the domain of messages and allows various constraints on the communication contents. The price to pay is the fact that we consider only finite prefixes or suffixes of the communication of the processes. Moreover, we focus mainly on synchronous communication, although some adjustments should allow for the analysis of asynchronous channels as well.

7.2 Domain

7.2.1 Challenges

Let us first recall the definition of the ith-prefix abstraction [14] (the ith-suffix is analogous). It is defined as a set D_i of words with length less than or equal to i. The neutral elements, $\oplus$ and the ordering are quite obvious:

- $\bar{0} = \emptyset$
- $\bar{1} = \{\epsilon\}$
- $\oplus = \cup$
- $\sqsubseteq = \subseteq$

The more interesting case is $\otimes$ which is defined

$$U \otimes V = \{(uv)_i \mid u \in U, v \in V\}$$

where $(w)_i$ is the prefix of w of length i. The already mentioned Galois connection is created by taking $v_a = \{a\}$ for every synchronization action a. It is important to emphasize here that the bound i refers to the number of communication actions of the whole system.

When we consider the problem of message passing, there are a few problems with the idea of using communicating pushdown systems and the ith-prefix/suffix abstractions. This is because the verification is done in two separate steps. First we use pushdown systems to analyze every process one at a time. And only then we use those results to reason about the whole system of communicating concurrent processes. However, this means that during the first step we have no information about the actual values being sent between the processes.

Since the values sent often influence the control flow, ignoring them leads to considerable imprecision. If the domain of those values is finite, we can generate a separate pushdown rule for each of the possibilities, reducing the problem to the approach from the previous section (as we have done in Chapter 6). However, this causes a blowup in the number of rules and is not always possible (e.g., in case of infinite domains). Furthermore, it can be often less precise, if the generated pushdown system does not maintain information about the contents of messages that have been sent in the past. Therefore, our solution to this problem is based on the idea of deferring reasoning about the contents of the communication to the second phase of analysis.

There is also another challenge — the ith-prefix/suffix abstractions are exponential in i and representing them efficiently is essential. However, in many programs the communication traces will have the same or similar prefixes and suffixes — this is due to conditional control flow, various loops, etc. Therefore, we introduce a data structure based on minimized DFAs to combat the explosion of the number of traces as well as allow efficient $\oplus$, $\otimes$ operators and equivalence checking.

7.2.2 Introducing constraints

In order to reason lazily about the values being sent between the processes we extend the domain of actions to include the variables from the program of interest. Moreover, since the control flow of a process is often based on the contents of messages, we also include constraints on the values of the variables. We define the set of possible actions Act as follows.

$$
\begin{array}{lcl}
\mathit{action} & ::= & a \mid b \\
a & ::= & (sr, ch, \vec{e}) \\
sr & ::= & \textbf{send} \mid \textbf{receive} \\
e & ::= & x \mid c \\
b & ::= & e_1 = e_2 \mid e_1 \neq e_2 \mid \textbf{true} \mid \textbf{false}
\end{array}
$$

where ch is a channel name, x is a variable name and c is a constant. Note that this is a basic variant of the language and more advanced constraints are certainly possible, such as arithmetic relations $\leq$ and $<$. We will expand on this point in one of the subsequent sections.

In the previous section we considered an abstract domain of the ith-prefix abstraction to be the powerset of traces with up to i communication actions. However, with the addition of conditions, the bound i on the number of communication actions is no longer enough to guarantee that our domain satisfies the *ascending chain condition* [59]. This is due to the addition of conditions — we can always continue adding new words that differ only in the order of conditions or in the number of repetitions of some conditions. But since the set of all possible conditions in a program is finite, it should be clear that at some point adding or reordering them does not bring any additional information. This motivates the following development, where we will make precise what traces provide some new information and which ones can be safely ignored.

We first introduce the notion of satisfiability of traces. Since the **receive** action can bind new values to existing variables, we cannot directly use the constraints

— consider for instance a trace:

$$(\text{receive}, ch, (x))(x = \mathsf{Foo})(\text{receive}, ch, (x))(x = \mathsf{Bar})$$

It should be satisfiable (x is bound to two potentially different values), even though when taking directly the conjunction of the conditions is obviously not. The solution is to introduce a fresh name every time a variable is bound to a potentially new value. This would correspond in our example to have two conditions: $x_0 = \mathsf{Foo}$ and $x_1 = \mathsf{Bar}$. We will discuss this approach in more detail in Section 7.4. For now we assume that we have a function Rename that performs the renaming of the variables whenever they are bound to new values.

Definition 7.1 A trace w of a PDS is satisfiable if and only if all the constraints of Rename(w) are jointly satisfiable.

Note that we are not taking into account the constraints due to communication and just focus on the problem of satisfiability of internal conditions of a given trace[1]. Now we are ready to define when traces are equivalent in terms of both the communication and satisfiability. For this, we introduce a morphism h_c defined as

$$h_c(a) = \begin{cases} a & \text{if } a \text{ is a communication action} \\ \epsilon & \text{otherwise} \end{cases}$$

Definition 7.2 We say that two traces t_1 and t_2 are equivalent if and only if they are equal under h_c (i.e., $h_c(t_1) = h_c(t_2)$) and are equisatisfiable (i.e., t_1 is satisfiable if and only if t_2 is).

Examples of equivalent traces would include traces that differ only in the order of conditions (in some consecutive series of conditions) or that repeat conditions (again without any communication action in between).

We will use the above to limit the number of traces that need to be considered. Let us first denote $\mathsf{Act_C} \subseteq \mathsf{Act}$ as the set of communication actions and $\mathsf{Act_B} \subseteq \mathsf{Act}$ as the set of condition actions, where $\mathsf{Act} = \mathsf{Act_C} \cup \mathsf{Act_B}$. With that it is simple to define a function that takes a word of our language of traces and collapses all subsequences consisting only of conditions into sets, which correspond to their conjunction:

$$\textsc{Collapse} : \mathsf{Act}^* \to (\mathsf{Act_C} \cup \mathcal{P}(\mathsf{Act_B}))^*$$

[1] However, the communication actions are still important since they can bind new values to the variables and thus are taken into account when renaming.

Lemma 7.3 *For every word w of Act^* we have that w and* $\textsc{Collapse}(w)$ *are equivalent.*

Proof. The function Collapse does not change the communication actions or their order, so $h_c(w) = h_c(\textsc{Collapse}(w))$. The only difference between w and $\textsc{Collapse}(w)$ is that in place of a sequence of conditions it creates a set of those conditions. The result then follows from the fact that conjunction is commutative and idempotent (i.e., the order of condition actions in a sequence of such actions is not important and $\forall a : a \wedge a \iff a$).

The immediate consequence here is that if we have two different but equivalent traces, we only need to consider one of them. Therefore, we refine our abstract domain to be $\bar{D}_i = \textsc{Collapse}(D_i)$.

Lemma 7.4 *For any program (and any i) under consideration the abstract domain $\bar{D}_i$ is finite.*

Proof. In the result of Collapse there cannot be two sets of conditions one after another — they would have been collapsed. Moreover, the number of communication actions is finite and any trace contains at most i such actions; similarly the set of all conditions is finite (and so is its powerset). So there is only a finite number of possible words in the language.

This immediately gives us that our domain satisfies the ascending chain condition and hence the Kleene iteration over constraints due to $Post^*$ and Pre^* algorithms, will terminate with the least solution. Furthermore, by considering equivalent traces, we do not lose any information with respect to global traces — there are no changes to the possible communication actions (ensured by equality under h_c) or conditions (ensured by equisatisfiability of conditions).

Finally, since we are extending the language of actions we also modify the definition of transition relation for CPDS. Since this development is quite simple we only mention the main differences compared to the original formulation. First of all, we require matching of "send" and "receive" actions. Furthermore, we want to only consider the global traces that satisfy all the constraints, which can be achieved by extending the notion of satisfiability to global traces. The only necessary addition is to generate the additional constraints due to the communication — if we have corresponding $(\textbf{receive}, ch, \vec{e_1})$ and $(\textbf{send}, ch, \vec{e_2})$ we require that $\vec{e_1}$ matches $\vec{e_2}$, which should give rise to simple equality constraints.

7.3 Data structure

7.3.1 Overview

Before we go into the details of the data structure that we use for representing the languages of traces, let us first describe the requirements that it must satisfy. In the following we will denote an instance of our data structure as d and the language it represents as $\mathcal{L}(d)$. For the $\oplus$ operator as well as the $\bar{0}$ and $\bar{1}$ elements the requirements are unchanged from the original i^{th}-prefix abstraction [14]. However, $\otimes$ must be slightly adjusted, namely we require that:

$$\mathcal{L}(d_1 \otimes d_2) = \{(w_1 w_2)_i \mid w_1 \in \mathcal{L}(d_1), w_2 \in \mathcal{L}(d_2)\}$$

where

$$(w)_i = \begin{cases} w & \text{if } w \text{ has less than } i \text{ communication actions} \\ u & \text{otherwise, where } u \text{ is the shortest prefix of } w \text{ that contains } i \text{ communication actions} \end{cases}$$

This is essential since otherwise we could have traces which are not precise up to i communication actions and our analysis would be unsound. Also note that instead of shortest prefix it is also possible to use the longest one.

One of the main challenges in using the ith-prefix abstraction is that in general the languages will be exponentially large in the i parameter. This is easy to see — if there are $|\mathsf{Act}|$ possible actions then there are $|\mathsf{Act}|^i$ possible words of length i using this alphabet. However, as already mentioned, we can hope that those languages will have a considerable amounts of repetition.

Therefore, we use a minimized DFAs which can often represent exponential number of traces quite compactly. Apart from the memory savings, there are some other advantages such as the fact that by the Myhill-Nerode theorem [41] a minimized DFA is unique (thus equivalence checking of two minimized DFAs is trivial). Furthermore, since $Post^*$ and Pre^* algorithms compute a weighted automaton, it is possible that there will be some redundancies across different weights (i.e., copies of the same languages). Our representation allows us to share the same structure in memory in those cases. Moreover, since we only deal with finite languages, all the DFAs will be acyclic. Therefore, we go a step further and use only one multi-DFA, where every state represents some language and a weight would simply refer to such state. We avoid the overhead of a separate minimization step by maintaining the invariant that there are no two different states whose corresponding languages are the same. This is inspired by multi-DFAs for fixed-length languages [25] and ROBDDs [12].

We will refer to our data structure as RDFA, which stands for Reduced DFA.

There is a body of existing work on minimized acyclic DFAs [13, 78, 79], however, most of it focuses on creating them from large and already existing dictionaries. Our situation is quite different since we construct the DFAs using two operators $\oplus$ and $\otimes$. The main challenge is the efficient implementation of the latter, in particular, the truncation of the language to the given length. We have considered two different ways of implementing the data structure:

- As a single pointer to a node in a multi-DFA that represents the language of interest. This is the most compact representation, but it does complicate the algorithm for $\otimes$ — a given node can be at different depths[2] depending on the path from the root. The difficulty stems from the fact we should not try to explore all the paths in the automaton as there might be an exponential number of them. A possible solution is to cache what nodes have been extended along with the depth of the current path.

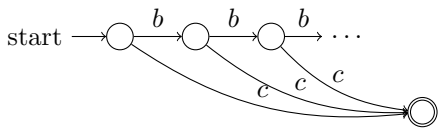
- As an array of i pointers to the nodes in multi-DFA, where each node represents a fixed-length language [25]. This representation might need some more memory than the previous one (we can not be able to share common sub-words due to the different number of communication actions). However, this representation does offer more information than the previous one and allows us to simplify the algorithm for $\otimes$. Furthermore, we believe that it offers more opportunities to further optimize the performance of the operations.

Because of the code simplicity and potential for optimization we have decided to use the second option.

Finally, we will review the problem of collapsing the transitions labeled with conditions. The reason for the introduction of this mechanism was to satisfy the ascending chain condition and thus ensure the termination of the Pre^* and $Post^*$ algorithms. Performing the collapsing for every two subsequent transitions labeled with conditions (i.e., where the target state of the first transition is the source for the second one) might seem excessive and sometimes leads to larger DFAs. As an example compare the two DFAs below, they represent the same language but the rightmost one corresponds to our rule of collapsing all subsequent conditions (a, b are conditions and c is a communication action):

[2] We use the word "depth" to describe the number of communication actions on the path from the root to the current node.

Therefore, it might be tempting to relax this requirement. However, even just limiting to collapsing states that have all in- and out-transitions labeled with conditions (instead of collapsing conditions themselves), make it possible for our DFAs to grow arbitrarily (b is a condition and c is a communication action):



Therefore, we have decided to impose the following invariant.

Definition 7.5 We say that an RDFA satisfies the condition simplification invariant if it does not contain a path with two subsequent transitions labeled with conditions.

Lemma 7.6 *For any given set of actions and value of i an RDFA satisfying the condition simplification invariant cannot grow arbitrarily.*

Proof. Since the condition simplification invariant amounts to collapsing every two subsequent conditions, then the language of the resulting automaton will directly correspond to an element from $\bar{D}_i$. Thus the result follows from Lemma 7.4.

It is worth mentioning that the collapsing is only necessary when performing the $\otimes$ operation. For the $\oplus$ if both sides already satisfy the invariant, then the result will satisfy it as well.

7.3.2 Details and algorithms

Recall that every state in a multi-DFA can be considered as representing some language, thus the mapping from languages to identifiers can be achieved in

a quite simple manner using a state signature (cf. [13]). We define a state signature as a pair consisting of a boolean value (indicating whether the state is final) and a set of out-transitions (i.e. pairs of labels and identifiers of the target state). It should be easy to see that if each state represents a different language, then the signature uniquely identifies it. Therefore, we use the idea of hash-consing [31] (another intuition behind this is that it is basically the same approach as taken in ROBDDs [12]). Thus, the main data structure is a hash table mapping the state signature (i.e., identifying a language represented by the state) to the state itself. We maintain the invariant that we never have two different states representing the same language — when creating a new node we always perform a lookup if a node with the same signature already exists and if so we use it instead of creating a new one. An important advantage of this approach is that equivalence checking is nothing more than testing the state identifiers for equality.

Apart from that, since our DFA is really a DAG (directed acyclic graph) and the nodes might be shared, all the operations use a copy-on-write approach (i.e., the nodes themselves are immutable). It is also important to be careful about implementing algorithms traversing the graph — we should always use memoization/dynamic programming to avoid traversing all the paths, since that would immediately make the algorithm exponential.

In the following we will often call the transitions labeled with a communication action as a-transition and one with conditions as b-transition. We keep them separate to simplify the implementation (and the presentation). Apart from that, we assume the existence of the following procedure:

$$\mathsf{NewNode} : \mathsf{Bool} \times \mathsf{ATransitions} \times \mathsf{BTransitions} \to \mathsf{Node}$$

that looks up (in a global hash table) if there is already an existing node with the same signature, if not it creates the new node and adds it to the hash table. The first argument specifies whether the resulting node should be final, the other two define its outgoing transitions. Since this is the only way of creating new nodes, we are guaranteed that no two nodes are identical. Finally, we will also assume the existence of a hash table with standard functionality.

7.3.3 Algorithm for $\oplus$

The algorithm for $\oplus$ amounts to creating a new node with the union of the out-transitions of the two arguments (calling itself recursively if there are two transitions with the same label). Since we maintain a cache with the results of all the recursive calls to $\oplus$ the worst case complexity of the procedure is

$\mathcal{O}(|d_1||d_2|)$ where $|d_1|$ and $|d_2|$ are the sizes of the DFAs, i.e., the number of states and transitions.

input : pointers lhs and rhs, combine-cache
output: pointer to the node representing lhs $\oplus$ rhs

```
if lhs = nil then
  | return rhs;
end
if rhs = nil then
  | return lhs;
end
if Find(combine-cache, (lhs, rhs)) ≠ nil then
  | return Find(combine-cache, (lhs, rhs));
end
map-a ← CombineTrans(OutTransA(lhs ), OutTransA(rhs ), combine-cache);
map-b ← CombineTrans(OutTransB(lhs ), OutTransB(rhs ), combine-cache);
return NewNode(IsFinal(lhs ) ∨ IsFinal(rhs ), map-a, map-b)
```

Procedure CombineNodes

input : maps lhs and rhs, combine-cache
output: map representing union of lhs and rhs

```
result ← lhs;
foreach (a, node) ∈ rhs do
  | if there exists a pair (a, node′) in result then
  |   | Erase(result, a);
  |   | Insert(result, a, CombineNodes(node, node′, combine-cache));
  | else
  |   | Insert(result, a, node);
  | end
end
return result
```

Procedure CombineTrans

Note that the algorithm works in the same way, no matter whether we consider the variant that includes the conditions or not. Additionally, we also should mark a node as final whenever one of the arguments is.

The structure of the algorithm is quite simple — we use two mutually recursive procedures to combine the nodes and combine the maps/sets of out-transitions. Since our RDFA representation is actually a sequence of pointers to the nodes, it calls the `CombineNodes` with the corresponding nodes (i.e., we combine nodes representing languages of k communication actions together for every $0 \leq k \leq i$).

input : arrays of pointers lhs and rhs, cache combine-cache
output: pointer to the node representing lhs $\oplus$ rhs

```
Initialize result array with nil (its size the same as lhs and rhs);
foreach index i do
  | result[i] ← CombineNodes(lhs[i], rhs[i], combine-cache);
end
return result
```

Procedure CombineRdfa

7.3.4 Algorithm for $\otimes$

The procedure for $\otimes$ is somewhat more complex. In the following we consider the computation of $d_1 \otimes d_2$. One of the main difficulties is, surprisingly, the fact that we need to perform truncation. Without it, we could simply insert the right transitions to each final state reachable from d_1. However, in our situation we need to be careful in order not to exceed the allowed bound. Also note that due to hash-consing the operation $d_1 \otimes d_2$ most often will create a "copy" of d_1 (since we are extending the language, the signature of most the nodes coming from d_1 will be different, thus we use a copy-on-write approach and create create new nodes for the resulting RDFA).

This has lead us into considering two different possibilities of interpreting the RDFAs and handling $\otimes$:

- Think about the RDFA nodes as representing a language of prefixes and thus $d_1 \otimes d_2$ should truncate nodes from the end of d_2 when creating the result of the operation. That would often require to create new nodes that represent truncated sub-languages of d_2 as well as a copy of d_1.
- Think about the RDFA nodes as representing a language of suffixes and thus $d_1 \otimes d_2$ should truncate nodes from the beginning of d_1 when creating the result of the operation.

The additional benefit of the latter option is that this leads to increased sharing of the d_2 subgraph. And since d_1 should be rebuilt anyway, we decided to use the second option. Note that it is still possible to get the language of prefixes — all one needs to do is to reverse $\otimes$ operator and then reverse the final RDFA. This may sound very expensive — in general reversing a DFA might lead to exponential blowup in the number of states (i.e., reversing the transitions gives rise to an NFA and determinization is in general exponential). However, our situation is somewhat special: we always work on minimized DFAs that

represent finite languages (i.e., the DFAs are acyclic) and we have not observed such a blowup. A similar situation with minimized, acyclic DFAs was considered in [79], where the author conjectures that in such special cases the reversal can be done in linear time.

The algorithm for $\otimes$ additionally uses a procedure to get a set of nodes reachable from the given one and representing the language truncated to the given length.

$$\textsc{Suffixes} : \mathsf{Node} \rightarrow \mathsf{Set\ Node}$$

Thus the main structure of the algorithm is to call the `ExtendNode` for each pair of nodes $d_1[k]$ and $d_2[l]$ for $0 \leq k \leq i$ and $0 \leq l \leq i$. Whenever $k + l > i$ we will have to extend the nodes from d_1 of height $i - l$ (in other words, we "drop" $(k + l) - i$ communication transitions). Each of the `ExtendNode` calls will visit each node of $d_1[k]$ at most once, since we memoize the results. Furthermore, at each such node we might have to perform `CombineNodes`. Finally, `ExtendNode` can potentially create an RDFA that violates the invariant of having two subsequent transitions labeled with conditions. Thus, we perform the collapsing of those conditions whenever necessary, which in the worst case might correspond to creating $\mathcal{O}(2^{|\mathsf{Act_B}|})$ transitions. This all amounts to the following worst-case complexity:

$$\mathcal{O}(\sum_{0 \leq k \leq i} \sum_{0 \leq l \leq i} 2^{|\mathsf{Act_B}|} |d_1[k]|^2 |d_2[l]|)$$

which can be simplified to:

$$\mathcal{O}(i^2 2^{|\mathsf{Act_B}|} |d_1|^2 |d_2|)$$

However, note that this is somewhat pessimistic bound, e.g., it would be surprising to create a system where at each node we need to call `CombineNodes` and where it exhibits the worst case complexity. Furthermore, quite often we will have the condition and communication actions interspersed (e.g., the control flow will be based on the results of communication), thus the collapsing might be less frequent. Finally, the bound does not take into account the fact that a final state can only be either a state without any out-transitions, or a state that has only transitions labeled with conditions that go to this last state.

For now our data structure and the corresponding algorithms build an ith-suffix abstraction (we are truncating the words from the left). Creating an ith-prefix abstraction is quite simple since our data structure forms a semiring. In a formal language semiring $a \otimes b = (a^R \otimes^R b^R)^R$, where $(-)^R$ corresponds to reversing the word and $a \otimes^R b = b \otimes a$. This is not quite the case in our i^{th}-suffix abstraction since we are truncating the strings from the left hand side. However, this suggests that we can reuse the `ExtendNode` algorithm and our

```
input  : pointers lhs and rhs, combine-cache, extend-cache
output: pointer to the node representing lhs ⊕ rhs

if lhs = nil ∨ rhs = nil then
  | return nil;
end
if Find(extend-cache, (lhs, rhs)) ≠ nil then
  | return Find(extend-cache, (lhs, rhs));
end
map-a ← ∅;
map-b ← ∅;
foreach (a, node) ∈ OutTransA(lhs ) do
  | node′ ← ExtendNode(node, rhs, combine-cache, extend-cache);
  | Insert(map-a, a, node′);
end
foreach (b, node) ∈ OutTransB(lhs ) do
  | if IsFinal(node) then
  |   | foreach (b′, node′) ∈ OutTransB(rhs ) do
  |   |   | InsertWith(CombineNodes, map-b, b ∪ b′, node′);
  |   | end
  |   | if OutTransA(rhs ) ≠ ∅ then
  |   |   | node″ ← NewNode(false, OutTransA(rhs ), []);
  |   |   | InsertWith(CombineNodes, map-b, b, node″);
  |   | end
  | else
  |   | node′ ← ExtendNode(node, rhs, combine-cache, extend-cache);
  |   | InsertWith(CombineNodes, map-b, b, node′);
  | end
end
result ← NewNode(false, map-a, map-b);
if IsFinal(lhs) then
  | result ← CombineNodesNotFinal(result, rhs, combine-cache);
  | /* Same as CombineNodes but the resulting state is never
  |    final.                                                  */
end
Insert(extend-cache, (lhs, rhs), result) ;
return result ;
```

Procedure ExtendNode

```
input  : arrays of pointers lhs and rhs, and two caches combine-cache,
         extend-cache
output: pointer to the node representing lhs ⊕ rhs

foreach index i of lhs do
    foreach index j of rhs do
        if i + j < N then
            lhs[i + j] = CombineNodes(lhs[i], ExtendNode(lhs[i], rhs[j]));
        else
            tmp ← Suffix(lhs[i], N − j − 1);
            foreach node in tmp do
                lhs[N − 1] ←
                CombineNodes(lhs[N − 1], ExtendNode(node, rhs[j]));
            end
        end
    end
end
```

Procedure ExtendRdfa

data structure to achieve i^{th}-prefix behavior. It is a simple matter of changing `ExtendNode(a, b)` to `ExtendNode(b′, a′)` where a' and b' are reversals of a and b respectively (this will drop elements from the front of reversed b, i.e., from the end of b).[3] But since a and b must have been created by modified `ExtendNode` then we do not actually need to perform the reversal. The only remaining thing to do is to reverse the final result before using it (e.g., presenting to the user).

Another interesting observation is that we expect the size of d_1 to be far more important for the efficiency of the algorithm than the size of d_2 (which hopefully in many cases will be shared). This has a quite interesting consequence with respect to Pre^* and $Post^*$ computations. The former generates mostly constraints of the form: $a \otimes x \leq y$ where x and y are variables and a is usually a single character word (corresponding to an abstraction of a single action). In other words, our abstract domain for i^{th}-suffix should perform quite well when solving such constraints. Similarly, in $Post^*$ computation we usually generate a lot of constraints of the form: $x \otimes a \sqsubseteq y$, i.e., exactly what we would expect to be less efficient. However, the discussion becomes more complex due to the fact that push-rules result in the constraints of the form: $x \otimes y \otimes a \sqsubseteq z$ ($Post^*$) or $a \otimes x \otimes y$ (Pre^*), which means that the actual performance will depend very much on the specific program (and the number of constraints of either type). Furthermore, the situation looks quite different once we use our data structure for i^{th}-prefix computations, since the order of arguments for `ExtendNode`

[3] We are ignoring here the additional arguments necessary for memoization.

is swapped. In Section 7.5 we will present some benchmarks that show the differences in performance depending on whether we use Pre^*, $Post^*$ and i^{th}-suffix or i^{th}-prefix.

7.4 Reachability

Once we have the $\mathcal{A}_{post^*}$ (or $\mathcal{A}_{pre^*}$) weighted automata for all processes (annotated with the state identifiers of the global RDFA), we can start reasoning about reachability of global configurations in our system. Since we use DFAs, one of the intuitive ways of checking their intersection could be the product construction of DFAs (done in a way that enforces synchronization on the right actions). However, in our case it is not clear how we could handle constraints due to message passing. What is more, we are not actually interested in the whole automaton resulting from the product construction — we only want to check whether there is at least one feasible trace. Therefore, we have decided to create a specialized search procedure that tries to find a trace exploring the DFAs in a depth-first manner. An additional advantage of this approach is that we can often terminate before exploring the whole state space.

Let us consider n DFAs (from n processes and the corresponding $Post^*$ or Pre^* computations). For the following we assume, without loss of generality, that the sets of variables used in each of the summarizations are disjoint. Now we can define the global state of the system of n processes as $\langle q_1 \cdots q_n, \sigma \rangle$, where σ is the set of constraints. Moreover, when exploring possible transitions, we use σ to check whether the current trace is feasible and update it with new constraints as we traverse the conditions or perform pattern matching of communication. We define our search in terms of two inference rules (for synchronization and condition actions respectively).

$$\frac{q_i \xrightarrow{a} q_i' \qquad q_j \xrightarrow{b} q_j' \qquad (c, \sigma') = \mathsf{match}(a, b)\sigma}{\langle q_1 \cdots q_i \cdots q_j \cdots q_n, \sigma \rangle \xrightarrow{c} \langle q_1 \cdots q_i' \cdots q_j' \cdots q_n, \sigma \cup \sigma' \rangle} \text{ if } \mathsf{sat}(\sigma \cup \sigma')$$

$$\frac{q_i \xrightarrow{a} q_i'}{\langle q_1 \cdots q_i \cdots q_n, \sigma \rangle \xrightarrow{a} \langle q_1 \cdots q_i' \cdots q_n, \sigma \cup \{a\} \rangle} \text{ if } \mathsf{sat}(\sigma \cup \{a\})$$

Whenever we perform matching or try to evaluate a condition we need to take into account the constraints. Note that when talking about the constraints we

are only interested in whether the trace is feasible, which in this case amounts to checking whether the constraints might have a solution.

Using those two rules we can explore the possible executions of the system and since our languages are finite, this process always terminates. We can terminate the search before exploring all possible states, if either of the two following conditions holds. The first one is finding a final global state, i.e., a state $\langle q_1 \dots q_n, \sigma \rangle$ where $q_i \in \mathsf{F}_i$ for all $0 < i \leq n$ and σ is satisfiable. The second one is performing i communication actions using the above rules. The latter one may be surprising, but it follows from the fact that we are only precise up to i communication actions. In both of these cases we have to conclude that the intersection might not be empty. Otherwise, we can conclude that the configurations of interest are unreachable.

It is interesting to note that having a specialized search procedure makes it possible to avoid the use of the shuffle operator and thus lifts the restriction that a channel is used by exactly two processes.

A natural choice for checking the satisfiability of the constraints during the search is to use a SAT or SMT (Satisfiability Modulo Theories) solver. However, before we can use such a solver for our purposes we need to solve the already mentioned problem of variable assignment (due to the **receive** action). This is not a new problem and has been encountered before [32, 38]. The solution employed there was to use the SSA (Static Single Assignment) [21] form for the trace or the program itself. Our situation is somewhat similar — even though the summarizations for all the processes represent sets of traces, at any time during the search we are considering only a single trace of the whole system. This makes it possible to perform the transformation on-the-fly during the search procedure and since any given trace is linear we can avoid using joining functions (often denoted as ϕ functions).

The same problem could also be caused by assignment to a variable other than through the **receive** action. Currently we simply disallow such a situation. A possible solution would be to extend the language of actions, however, since it would not count to the limit i, one must be careful to ensure the termination. Another possibility would be to avoid using such a variable in the constraints captured by the abstract domain.

7.5 Experiments

We have implemented our abstract domain in C++ and integrated it with our pushdown systems library (we will talk about it more in Chapter 8). In this section we will have a quick look at a few interesting benchmarks and performance comparisons. We will mainly look at constraints corresponding to computing the $Post^*$ of the entry of the `main` procedure or the Pre^* of its exit point.

First, we will consider a simple program that nicely presents the exponential nature of the problem:

```
proc main
  if
    :: true => a?(x)
    :: true => b?(x)
    :: true => c?(x)
  fi;
  if
    :: x < 0     => call main
    :: !(x < 0) => skip
  fi;
  d!()
end
```

When looking just at the synchronization actions the program corresponds to the words of the form $(a|b|c)^n d^n$ where $n \in \mathbb{N}$. In our case it is slightly more complex since we also capture the conditions.

In the Table 7.1 we use $Post^*$ to generate a set of constraints and present some statistics about the part of RDFA representing the weight summarizing the whole program (i.e., going from the entry to the exit point of the `main` procedure).[4] This should clearly present that explicit representation of all the traces is not really a feasible approach.

For comparison the Figure 7.1 presents the time of solving the constraints arising from running the $Post^*$ and Pre^* algorithm. Note that the time is not linear in the size of the i parameter (which is to be expected from our complexity analysis). However, it is growing far slower than the number of traces we represent.

Now let us also consider examples illustrating the difference in performance of

[4] We are presenting the result in form of a table instead of a graph due to the fact that the blowup in the number of traces is so significant that a graph would be simply unreadable.

	Nodes	A-Transitions	B-Transitions	Traces
5	15	18	6	17
10	53	62	19	242
15	110	138	41	4373
20	183	227	64	59048
25	280	358	101	1062881
30	388	492	134	14348906

Table 7.1: Statistics for the summarization of the first example.

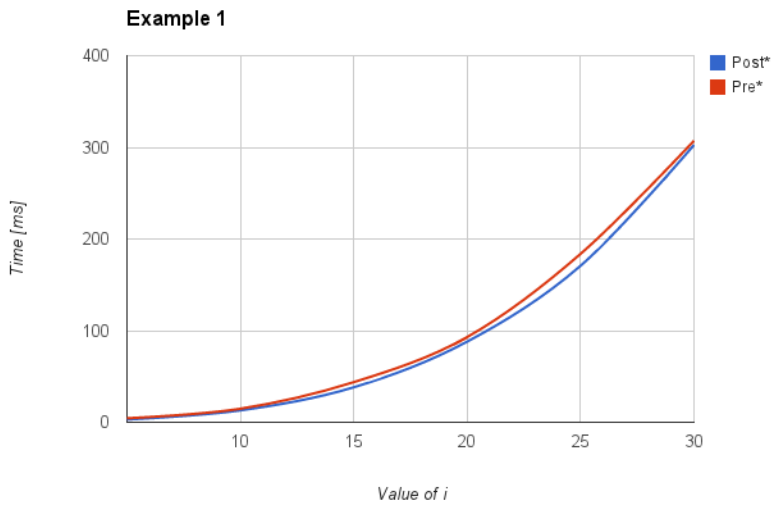


Figure 7.1: Performance of the domain for the first example.

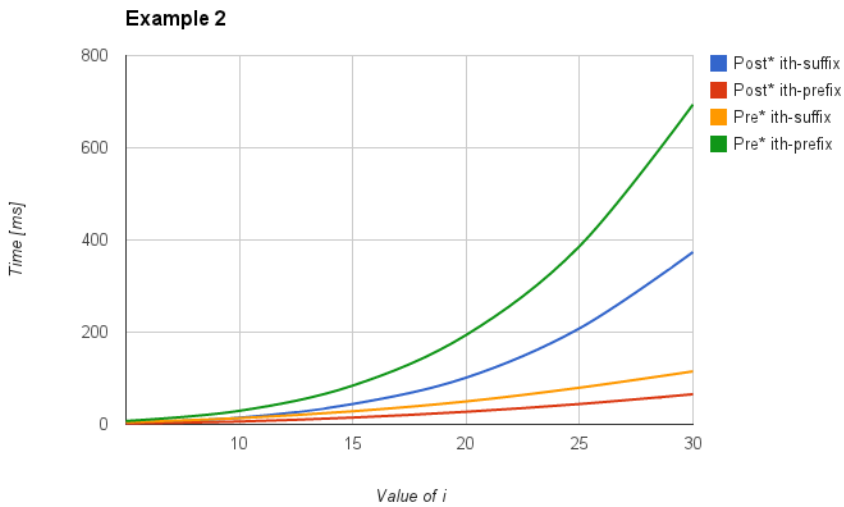


Figure 7.2: Performance of the domain for the second example.

Post* and *Pre** depending on whether we use the ith-prefix or i^{th}-suffix abstraction. In the first one, we have a loop that is the source of almost all traces:

```
proc main
  a?(x);
  do
     :: x < 0 => a!()
     :: 0 < x => b!()
  od
end
```

The graph in Figure 7.2 presents the solving time against the length of i parameter. As we expected in the previous section, the shape of the constraints makes the ith-suffix for *Pre** and i^{th}-prefix for *Post** much faster than their counterparts.

However, the picture looks slightly different if we consider a program close to the first example (we simply remove the final send action over channel d).

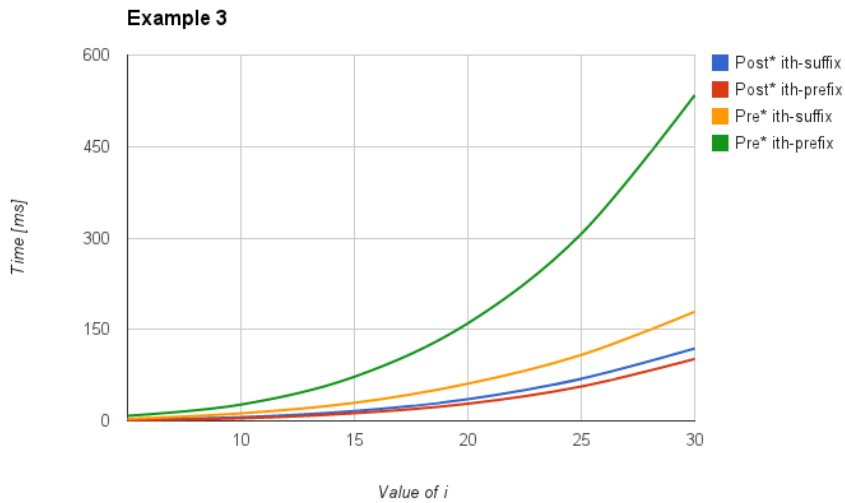


Figure 7.3: Performance of the domain for the third example.

```
proc main
  if
    :: true => a?(x)
    :: true => b?(x)
    :: true => c?(x)
  fi;
  if
    :: x < 0     => call main
    :: !(x < 0) => skip
  fi
end
```

The time required to solve the equation system is presented in the Figure 7.3. In this situation there is almost no difference between the ith-prefix and i^{th}-suffix for $Post^*$ and the constraints arising from Pre^* require more time to solve.

Our conjecture is that whenever a program has a very high number of procedure calls compared to ordinary statements, then the ith-prefix for $Post^*$ might in fact be very close to the ith-suffix (or maybe even better). However, one should note that situation is usually not very common — most programs have far more ordinary statements than procedure calls, and in those cases i^{th}-prefix should

perform better in $Post^*$ computation than i^{th}-suffix. Interestingly the results for Pre^* are quite similar to the previous example. Our conjecture is that the constraints corresponding to push-rules are slightly different than in case of $Post^*$ — there are two variables on the left-hand side that correspond to "large" RDFAs, whereas in the $Post^*$ one of them often corresponds to some smaller language. Thus reordering the operations might result in a more significant difference in this case.

There are still a number of optimizations that may be performed. For instance, introducing caching between calls to `ExtendNode` or `CombineNodes` might be beneficial. Furthermore, finding the suffixes in `ExtendRdfa` could be cached as well. Apart from that, there is a number of micro-optimizations that could improve the performance of the domain (more specialized data structures, less allocation in the inner loops, etc.). We also note that our algorithm for fixed-point computation could be significantly improved — it would not change most of the above comparisons, but should be able to offer some speedup. Some of those opportunities include iterating over the reverse post-order of the constraint graph, or computing the strongly-connected components of the graph, etc. [59].

7.6 Conclusions

Message passing (with all its variants) is one of the main approaches used for concurrency. Therefore, the ability to analyze and verify systems using message passing is increasingly important. This chapter continues the line of work on using communicating pushdown systems for verification of recursive processes. We have extended the ith-prefix/suffix abstraction with symbolic encoding of messages along with various constraints imposed on them. This makes it more efficient than the straightforward adaptation to message passing and allows to handle systems where the domain of messages is infinite. Furthermore, the inclusion of constraints enables us to express that some communication can happen only if certain conditions (e.g., referring to earlier communication) are satisfied. The domain also admits a very flexible way of adjusting the precision and the performance of the analysis. Using a SAT/SMT solver to handle the constraints means that we can use various combinations of theories, which gives some flexibility with respect to the domain of messages as well as the possible constraints. By combining the domain with the communicating pushdown systems we gain the ability to analyze recursive programs and easily obtain analysis results for various queries based on the stack contents of the given process.

In the future we would like to perform more experiments and invest more time into evaluating the scalability of our implementation. Moreover, the current

approach, even though it admits variables over infinite domains, is still considering only finite languages of communication actions. Thus, another line of work could be focused on trying to encode some constraints over the contents of the messages while considering more than finite languages.

Chapter 8

Library for pushdown systems

In this chapter we will present a new approach to the implementation of Pre^* and $Post^*$ algorithms as well as a new C++ library for algebraic analysis based on weighted pushdown systems. The new algorithms allow for computation of $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata without transforming the program representation to pushdown rules — instead they are adapted to work on a graph similar to the ordinary program graphs often used for static analysis. The library itself includes a number of improvements over other similar projects including the ability to create symbolic constraint systems and thus allowing the use of various constraint solvers (e.g., the ones based on Kleene iteration or Newton's method).

The structure of the chapter is as follows. First of all, in Section 8.1 we will show the motivation behind the need for a new library and discuss the its main objectives. Then, Section 8.2 will present the architecture of the library as well as some of its design decisions. In Section 8.3 we will focus on describing and discussing the main ideas behind the graph-based algorithms for Pre^* and $Post^*$. We evaluate our library in Section 8.4 discussing its main advantages as well as comparing its performance to a well-known library for WPDS. Finally, we conclude in Section 8.5.

8.1 Introduction

Pushdown systems and weighted pushdown systems have been already implemented and used in various tools. For this reason starting a new implementation for pushdown systems might seem unnecessary. Initially we shared that view and used a well-known Weighted Automata Library (WALi) [48]. However, it turned out that it was simply not flexible enough for our purposes. First of all, WALi is built around the algorithms presented in [66, 67] which combine the saturation procedure for constructing the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata with the least fixed-point computation of the weights corresponding to the transitions of the automata. Thus it requires that the abstract domain satisfies the ascending chain condition.[1] This design decision has some profound consequences, for instance, it makes it difficult to use different constraint solvers, e.g., based on the Newton's method [27]. Furthermore, it makes it necessary to equip every pushdown rule with a concrete weight from the abstract domain. This makes it difficult to share the structure of the pushdown system between analyses over different abstract domains. Finally, the WALi library is not thread-safe — it is not safe to perform multiple Pre^* or $Post^*$ computations at the same time. In many cases this is not a significant limitation, but in case of CPDS, which are quite natural to parallelize, it seems wasteful to not be able to fully utilize today's multi-core systems.

Based on those main issues we have decided to create a new library that would lift all of the above limitations and provide us with more flexibility to experiment. The main goals of our implementation are characterized below.

Modular: To be able to lift the above described restrictions and allow the user more flexibility in how he/she uses the library, it should be very modular. For instance, we want to allow using different types of constraint systems as well as various constraint solvers. Furthermore, it should be possible to use one pushdown system (i.e., one model of a program) and use it for performing different Pre^* and $Post^*$ computations.

Thread-safe: Most of modern computers have at least dual-core CPUs with quad-cores becoming more and more popular. Our library should be safe to use in a multi-threaded environment. This is especially interesting for CPDS since the analysis stage performs the $Pre^*/Post^*$ computations and constraint solving independently for each process.

Efficient: Performance of the library is an important consideration since pushdown systems are often used for whole program analysis. Thus the size of

[1] Or at least that the Kleene iteration terminates — it should be possible to use some form of a widening operator in place of $\oplus$.

the problems might be quite large and good scalability is essential.

We believe that our implementation accomplishes all the above goals. In the next section we will present more details about the architecture and design choices of the library as well as the algorithms it is based on.

8.2 Architecture

The library consists of three main components:

Pushdown system: The class representing a pushdown system contains some representation of all the pushdown rules. It provides the ability to add new pushdown rules, as well as perform the Pre^* and $Post^*$ computations that produce a constraint system and a $\mathcal{A}_{pre^*}$ or $\mathcal{A}_{post^*}$ automaton.

Constraint system: Represents the constraints generated by Pre^* or $Post^*$ computations. Since we might have various kinds of constraints (e.g., commutative, non-idempotent, etc.) we allow the user to supply the specialized data structure for constraint system.

Constraint solver: Since we want to allow various constraint solvers (including ones where the standard Kleene iteration does not terminate in a finite number of steps), the user is free to supply the constraint solver to be used.

Let us discuss the first component, namely the pushdown systems. One of the main factors that influenced its design was the internal representation of pushdown rules/pushdown system. Most libraries for pushdown systems operate on the pushdown rules and thus require the user to supply a set of such rules. However, in many situations the program is already represented available as, e.g., program graph. So converting it to a set of pushdown rules amounts to duplicating the information about its structure.

We wanted to avoid this problem and one of the main design decisions is that all the pushdown rules are stored as a graph. The library is polymorphic in the type of the graph and so the user is able to supply his/her own graph type (as long as it satisfies a few requirements). This makes it possible to use existing program graphs/flow graphs with the library. We believe that this is an important contribution. By making it possible to use a program graph directly to represent the pushdown system, we are able to not only save space due to avoiding the duplication, but we also save time that would be spent on creating

the pushdown system (since the graph is already available). Furthermore, this makes it easier for existing projects to experiment and use the library.

Of course, this graph representation would not be very useful if the Pre^* and $Post^*$ algorithms could not efficiently use it for computation. Fortunately it is not only possible but also quite efficient to do so. In Section 8.3 we present a new approach to the Pre^* and $Post^*$ algorithms that make it possible to perform the computations directly on the pushdown graph. They are worklist-based algorithms that intuitively can be thought of as backward (in case of Pre^*) or forward (in case of $Post^*$) graph traversals.

Another important consideration is that we want to be able to use the same pushdown system for different analyses. Therefore, Pre^* and $Post^*$ computations do not modify anything in the data structures representing the pushdown system (i.e., they only have "read-access" to it). This immediately makes it possible to run multiple concurrent Pre^* and $Post^*$ computations using the same pushdown system. Apart from that, both the Pre^* and $Post^*$ procedures create a symbolic constraint system. Instead of using concrete weights we use symbolic identifiers of the corresponding pushdown edge or a special constant such as $\bar{1}$. The advantage of this solution is that it is possible to run Pre^* or $Post^*$ to create the constraint system without deciding what analyses to actually run. The user is then able to reuse the already computed constraint system over various abstract domains. It is possible, for instance, to compute first a result of a less precise analysis and only resort to a more precise one if the first was not enough to prove the property of interest. In existing libraries like WALi, it would be necessary to either compute both of them in the first place, or run Pre^* or $Post^*$ twice.

This brings us to the constraint solvers. Since the constraint system can be concurrently used by different solvers, they all must treat it as immutable. This again allows to utilize multiple threads, without any overhead of synchronization. We provide a standard least fixed-point solver based on Kleene iteration. But it is also possible to use solvers such as the one presented in the Chapter 9 that is based on the Newton's method [27].

Finally, it is important to note that there is a cost to be paid for the modularity and thread-safety. In a few places, the fact that both Pre^* and $Post^*$ treat the pushdown system in a "read-only" manner means that some of the information that could be stored directly in the pushdown graph, must be stored in some separate data structures (we use mainly hash tables for that purpose). Similarly since the constraint systems should support different abstract domains and solvers, they represent the constants symbolically and cannot include the concrete values from the domain. However, we believe that this is a small price to pay for the additional flexibility and the ability to take advantage of mod-

ern hardware. Fortunately, as we will see in Section 8.4 the performance hit is acceptable and we believe well worth the additional benefits of the library.

8.3 Graph-based algorithms

When software verification first started using techniques based on pushdown systems, they were mainly applied to boolean programs. The control locations are used to model the valuation of global variables and stack alphabet to model the program location (and return addresses of procedures), as well as the valuation of local variables. In this setting it is often impractical to use explicit representation of pushdown rules and thus symbolic techniques (based on BDDs [12]) were developed in [72]. However, the approach is not really suitable when one wants to go outside of finite abstractions (recall control locations and stack alphabet must be always finite). This is one of the main reasons for the weighted pushdown systems, which annotate each rule with a semiring value. This allows for using a wide variety of abstract domains including the ones that are not finite. One of the interesting consequences is that when using WPDS it is quite common to have only one control location and express all the dataflow information in the abstract domain. More importantly, this makes the two representations of the program, namely as a program graph or a set of pushdown rules, very similar to each other. This realization is one of the motivations for our new approach to the Pre^* and $Post^*$ algorithms.

Let us first consider an example of the above observation using the following example:[2]

```
proc main
  a?(x);
  call f;
  b!(x)
end

proc f
  if
    :: x < 0     => call main
    :: !(x < 0) => skip
  fi
end
```

[2]We have created a simple front-end for a variant of NanoPromela [3] extended with procedure calls.

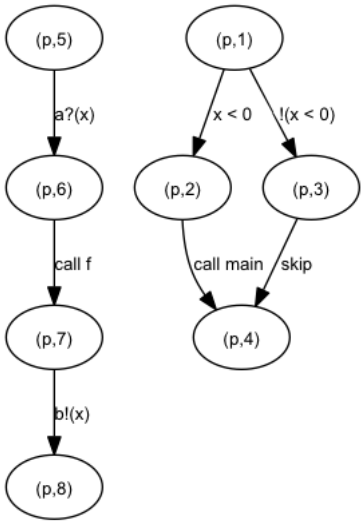


Figure 8.1: Program graph corresponding to the example.

We have depicted its corresponding program graph on Figure 8.1.

In order to use WPDS techniques, we could create the following pushdown rules.

$$\langle p, 1\rangle \xhookrightarrow{\texttt{x < 0}} \langle p, 2\rangle$$
$$\langle p, 1\rangle \xhookrightarrow{\texttt{!(x < 0)}} \langle p, 3\rangle$$
$$\langle p, 2\rangle \xhookrightarrow{\texttt{call main}} \langle p, 5, 4\rangle$$
$$\langle p, 3\rangle \xhookrightarrow{\texttt{skip}} \langle p, 4\rangle$$
$$\langle p, 4\rangle \hookrightarrow \langle p, \epsilon\rangle$$

$$\langle p, 5\rangle \xhookrightarrow{\texttt{a?(x)}} \langle p, 6\rangle$$
$$\langle p, 6\rangle \xhookrightarrow{\texttt{call f}} \langle p, 1, 7\rangle$$
$$\langle p, 7\rangle \xhookrightarrow{\texttt{b!(x)}} \langle p, 8\rangle$$
$$\langle p, 8\rangle \hookrightarrow \langle p, \epsilon\rangle$$

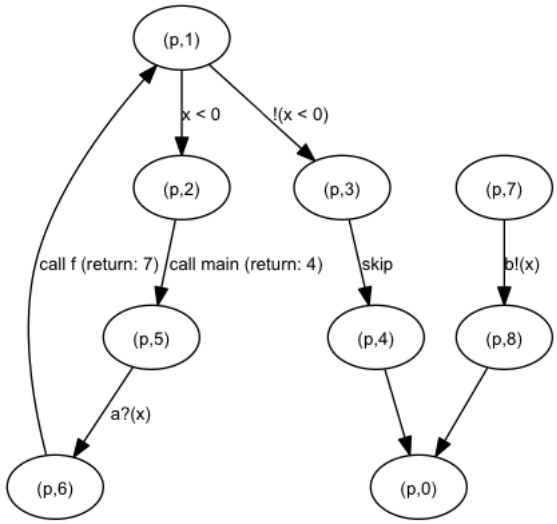


Figure 8.2: Pushdown graph corresponding to the example.

Note that these are really nothing else than the set of edges of the above program graph (with some additional information, such as return addresses). This motivated us to try and reformulate the Pre^* and $Post^*$ algorithms in terms of graph traversal, in order to reuse the existing program representation. As we will see below, it is not only possible to do that but also makes efficient implementation much more straightforward.

First of all, let us introduce the notion of a *pushdown graph* that represents all the pushdown rules. In general a node in this graph is identified by an element from $P \times (\Gamma \cup \{\epsilon\})$. Since we also need to handle the push rules (which have two stack elements on the right-hand side), we additionally have that an edge in the graph can be labeled with a stack element (e.g., the return address). For instance, if there is a push rule $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3 \rangle$ then we will have an edge from (p_1, γ_1) to (p_2, γ_2) that is labeled with γ_3. Clearly with this "encoding" into a graph form we do not lose any information. We present the pushdown graph for the example in Figure 8.2. Of course, it is essential to create Pre^* and $Post^*$ algorithms that are able to use this representation directly (otherwise there would be little benefit from it).

Furthermore, note that most existing flow graphs can be interpreted in this manner without any modifications. In other words, it is often possible to use the original program graph and simply interpret it in a slightly different way. The main difference is in the handling of call-edges, which in a program graph

go to the return point (and are labeled with the call to the right procedure), whereas in a pushdown graph, to the entry point of the corresponding procedure (and are labeled with the return point). Therefore, all the information is already available in the program graph and in treating the call-edges in a special way, we obtain a pushdown graph. This obviously makes it possible to reuse the existing representation of the program and use it efficiently for analysis using techniques based on pushdown systems.

Below we will describe our Pre^* and $Post^*$ algorithms that are capable of working directly on the pushdown graphs. For clarity of presentation, for now we will ignore the problem of generating the constraint system. This is a straightforward extension (we simply add a corresponding constraint when trying to add a transition) and is not really different from the usual approach taken by CPDS or WPDS.

8.3.1 *Pre**

Let us first consider the case of Pre^*. Recall that the saturation procedure can be characterized by the following rule:

> If $\langle p_1, \gamma\rangle \hookrightarrow \langle p_2, w\rangle \in \Delta$ and $p_2 \xrightarrow{w}{}^* q$ in the current automaton (for some $q \in Q$) then add a transition $p_1 \xrightarrow{\gamma} q$.

It should be clear from the above that adding one transition will often make it possible to apply the above rule with other pushdown rules and thus add some more transitions. For instance, if we have $p_3 \xrightarrow{\gamma_3} q$ in the automaton and:

$$\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\rangle \hookrightarrow \langle p_3, \gamma_3\rangle$$

then we will first add a transition $p_2 \xrightarrow{\gamma_2} q$. This in turn will make it possible to add $p_1 \xrightarrow{\gamma_1} q$. If we think about the pushdown system as a graph, we simply perform a backwards traversal and add transitions at each visited node. The only challenge are the push-rules, which need a bit special handling — this is the only situation where we need to check for existence of a path instead of a single transition. Since it is easier to present the problem with an example, consider the following rule:

$$\langle p_1, \gamma_1\rangle \hookrightarrow \langle p_2, \gamma_2\gamma_3\rangle$$

which will be represented as a following edge in the pushdown graph:

$$(p_1, \gamma_1) \xrightarrow{\gamma_3} (p_2, \gamma_2)$$

If the traversal is at the right node, i.e., (p_2, γ_2) a natural thing to do would be to insert the transition $p_1 \xrightarrow{\gamma_1} q_2$ for every q_2 such that there exists: $p_2 \xrightarrow{\gamma_2} q_1 \xrightarrow{\gamma_3} q_2$ and continue with the traversal from (p_1, γ_1) with q_2 being the new target. However, this can cause us to miss some transitions. If the transition $q_1 \xrightarrow{\gamma_3} q$ (for some q) is added after visiting (p_2, γ_2) we might miss adding $p_1 \xrightarrow{\gamma_1} q$. At the same time, for performance reasons we would prefer to avoid traversing the graph multiple times. To solve the problem, we simply add some additional information to the intermediate node (in this case q_1), namely that there is a "suspended traversal" that should be resumed once a transition with the given alphabet element (in this case γ_3) is added to the intermediate state. Once we add such a transition, we should examine all such suspended traversals. This leads to a worklist-based algorithm where we keep track of the current edge in the pushdown graph as well as the current target state in the automaton (i.e., the target state of the transitions that we are adding). Finally, it is nice to note that if the transition we are about to add has already been added, then we can simply stop the traversal and continue with the next worklist element (i.e., since we already added the transition before, we must have also added the relevant worklist elements as well).

Having the high-level description, let us have a look at some of the details of the algorithm. It works in three stages:

1. Populating the worklist based on the initial automaton.
2. Handling the pop-rules.
3. The main worklist-based graph-traversal loop.

The first stage examines transitions of the initial automaton and based on that initializes the worklist. We only need to consider the outgoing transitions of the initial states, lookup the corresponding node in the pushdown graph and push all the in-edges[3] along with the target state of the automaton transition into the worklist. If any of the edges in the pushdown graph represents a push rule, we additionally need to check if the automaton contains another transition matching the return location and only then add it to the worklist.

Handling of pop-rules is quite straightforward. We need to find all the vertices in the pushdown graph that correspond to a pair in $P \times \{\epsilon\}$ and examine all their predecessors adding the corresponding self-loops on every initial state. Of course after each addition of a new transition, we should update the worklist as above.

[3]Since in Pre^* we perform backwards traversal.

The third stage is the most interesting and complex — it expresses the main saturation procedure. We proceed as long as the worklist is not empty and at each step:

1. We pop a worklist element consisting of an edge e in the pushdown graph and a target state s (of the automaton).
2. We look at source node of the edge e with the corresponding control location p and stack alphabet γ and try to add $p \xrightarrow{\gamma} s$ to the automaton.
3. If the transition already exists we continue with the next worklist element. Otherwise, we check if the current source of the added transition has some suspended worklist elements and if so we add them to the worklist (of course checking if the alphabet element matches).
4. Finally, we look for all the predecessors of the source node of the current edge.
   - If the predecessor's edge is not a push rule we simply add it to the worklist with the same target state.
   - If it is a push rule, then we add it to the suspended list of the current target state. If additionally there is a corresponding path in the automaton, we can add the relevant elements to the worklist.

The pseudocode for the algorithm is available in Appendix C.1.

8.3.2 *Post**

Some of the main ideas behind the graph-based algorithm for *Pre** are also applicable to the case of *Post** computation. First, let us recall the saturation procedure for *Post**.

For every pushdown rule r in Δ:

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \epsilon \rangle$ and there is a path $\rho = q \overset{\gamma}{\dashleftarrow} p$ then add a transition

$$q \xleftarrow{\epsilon} p'$$

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and there is a path $\rho = q \overset{\gamma}{\dashleftarrow} p$ then add a transition

$$q \xleftarrow{\gamma'} p'$$

- if $r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and there is a path $\rho = q \overset{\gamma}{\dashleftarrow} p$ then add transitions

$$q \xleftarrow{\gamma''} q_{p',\gamma'} \qquad q_{p',\gamma'} \xleftarrow{\gamma'} p'$$

The most fundamental difference compared to Pre^* is that, in terms of pushdown graph, we are doing a forward traversal. Another quite obvious difference is the fact that we have ϵ-transitions in the automaton (not in the initial one, but we add them during the saturation procedure and must take them into account while checking for existence of paths). However, the basic structure of the algorithm remains the same — we make use of a worklist of the current pushdown edge (that corresponds to a pushdown rule) and a state in the automaton that will be a target for the added transition(s). Since we only check for existence of the left-hand side of pushdown edges and they always have just one stack element, we do not need any notion of suspended traversals that were necessary in Pre^* case for matching paths of two transitions in case of push rules. On the other hand, to efficiently handle the ϵ-transitions (i.e., we do not want to check and follow ϵ-transitions all the time) for each state we keep a set of states that are its ϵ-predecessors (i.e., states from where one can make an ϵ-transition to the state in question).

The first stage of the algorithm is almost the same as in the case of Pre^* — we initialize the worklist based on the transitions existing in the initial automaton. Then we go on to the main main saturation loop — we pop a pushdown edge e and the corresponding target state s from the worklist and have two choices on how to proceed.

The first case is when the edge e is not labeled with a return address (i.e., the corresponding rule is not a push-rule and has either zero or one stack element on the right-hand side). We then proceed as follows:

- We examine the target node of the edge e and its corresponding control location p and an element of the stack alphabet γ_ϵ and try to add the transition $p \xrightarrow{\gamma_\epsilon} s$.
- If it has already been added, we continue with the next worklist element.
- Otherwise, if the added transition is not an ϵ-transition we simply update the worklist with all the pushdown edges originating at node corresponding to the (p, γ_ϵ) pair. If we have added an ϵ-transition, we also add the source state of the newly added transition to the set of ϵ-predecessors of target state. Furthermore, we add to the worklist all edges/rules that might fire due to the out-transitions of the target state (of our newly added transition).

The second case is when the edge e is labeled with a return stack element (i.e., it corresponds to a push-rule). This situation is slightly more complex because we are adding two transitions (neither is an ϵ-transition) and potentially a new state to the automaton. Let the target of the edge e correspond to some control locations p and stack element γ_1, while e itself is labeled with γ_2. We try to add the state q_{p,γ_1} and two transitions $p \xrightarrow{\gamma_1} q_{p,\gamma_1}$ and $q_{p,\gamma_1} \xrightarrow{\gamma_2} s$. Therefore, when updating the worklist we need to look at transitions outgoing from both the state p and the intermediate state q_{p,γ_1}. This is essential, because there might be an ϵ-transition into the latter state and the worklist must be populated with all edges/rules that might applicable due to that (at this place we use the set of ϵ-predecessors stored for each intermediate state).

The algorithm stops once the worklist is empty — which is inevitable since there is only a finite number of transitions that can be added and we detect the situation of already existing transition.

The pseudocode for the algorithm is available in Appendix C.1.

8.4 Discussion

In this section we will briefly discuss the main advantages and contributions of our graph-based saturation algorithms as well as its implementation. We believe that one of the most important advantages is the ability to reuse the representation of a program. Most other implementations of the saturation procedures, require the user to convert the program into a set of pushdown rules. In real applications this might be wasteful — all the information about the program might have to be duplicated. Our graph-based approach is capable of using some of the possible representations of a program directly. Of course, there are some restrictions, but most program graphs will have enough information to be used in this setting without any modifications. The significance of that is more pronounced if we consider that for good results one often wants to perform whole-program analysis when using pushdown systems, which results in quite large systems. Furthermore, a pushdown graph should be quite efficient, since the main saturation procedure is actually very close to a graph traversal. Some of the other approaches store this information in additional data structures, such as a hash table[4]. This results in higher constant factors due to cache locality as well as additional overhead of maintaining a hash table (or two if we want to run both Pre^* and $Post^*$ algorithms) with an entry for each pushdown rule. Thus, we believe that even though the approach does not provide an asymptotic

[4]WALi [48] takes this approach.

improvement, it is an important contribution that makes implementation of the technique in real world software more likely and more efficient.

Other important contributions include the details of our implementation. We believe that the most important ones are:

- Modularity — building an automaton is separated from the constraint generation. This might seem at first to be a minor advantage, however, it makes it possible to use separate constraint solvers such as the one presented in Chapter 9. This is simply not possible with the approach taken by WPDS [66, 67] and WALi library [48]. Therefore, this small improvement results in our ability to use some abstract domains that simply cannot be handled otherwise.
- Symbolic constraint systems. This allows to create a constraint system for the given pushdown graph once and calculate the solutions for various analyses (using different constraint solvers) without repeating the saturation procedure. Combined with thread safety it allows one to perform the analyses in parallel.
- Thread-safety — we believe that it is an important advantage when discussing analyses based on CPDS, since at first the analysis problems of different threads are completely independent and thus can be performed at the same time.
- Polymorphism allows the library to be used by other software tools more easily. One of the examples is that the library will work with any type of graphs, as long as it satisfies some basic requirements. Similarly it allows the user to specify the types of the data structures used to identify the control locations, stack alphabet, etc.

One of the more important questions to ask is how much the additional flexibility will hurt the actual performance of the algorithms. To test that we have created a synthetic benchmarks and generated pushdown graphs using both our implementation of graph-based algorithms and WALi [48]. For that we have generated a few programs of various sizes that would allow us to perform some preliminary assessment of the performance and scalability of our library (in Appendix C we give a short description on the generated programs). Our library separates the constraint solver from the automata construction thus measuring only the time spent on the latter is quite straightforward. However, we do include the time to create the constraint system in the benchmark. In case of WALi (which combines the two) we have created a trivial abstract domain that should not have any influence on the benchmarks. We have created two scenarios far WALi benchmark — one that measures just the Pre^* and $Post^*$

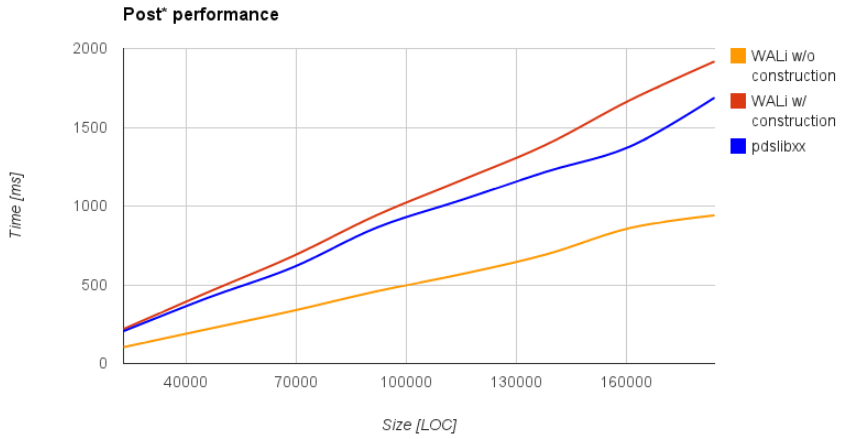


Figure 8.3: Performance of $Post^*$.

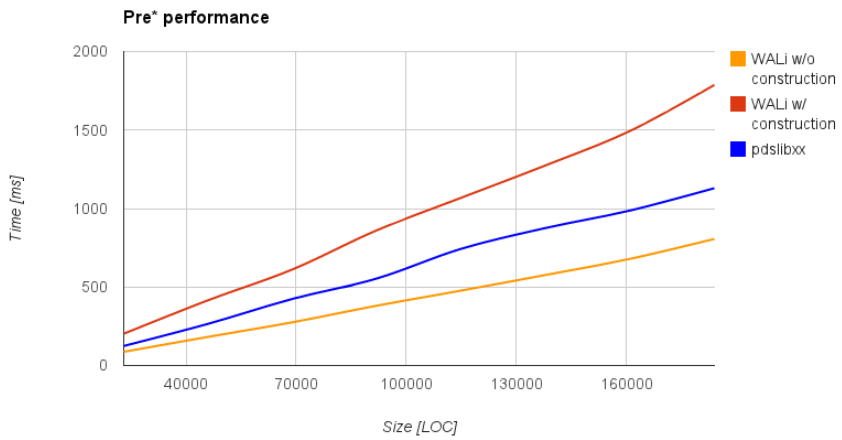


Figure 8.4: Performance of Pre^*.

computations and a second one that additionally includes the time to create the pushdown system. Note that the second scenario includes our code to traverse the program and add the rules, thus it should not be considered as a benchmark of WALi itself. However, we do believe that it gives an additional insight into the cost of recreating the program in the form of a set of pushdown rules.

The results are presented in Figure 8.3 for $Post^*$ and in Figure 8.4 for Pre^*. As we can see WALi slightly outperforms our implementation in the pure Pre^* and $Post^*$ computations. But once we also factor in the cost of creating the pushdown system, our library performs better — as we could expect in situations when the program graph is already available the ability to use it directly is quite beneficial. Furthermore, we believe that there is still a number of possible optimizations that we could perform to make our library more efficient (e.g., using specialized data structures, avoiding some of the allocation, various other micro-optimizations, etc.)

8.5 Conclusions

In this chapter we have presented a new approach to thinking about and implementing Pre^* and $Post^*$ algorithms. Some of the ideas (e.g., using hash tables to match a pair of control location and stack element to match rules that have it on right-hand side or left-hand side) have been already present in [48]. However, we are advocating a different way of approaching the whole problem that is based on reusing the internal representation of a program and making the algorithms work directly on the program graphs, without the need to convert them to sets of pushdown rules. We believe that this contributes to making the technique easier to use and integrate into existing tools. Moreover, separating the constraint solving from computing the $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ automata opens up possibilities of performing new analyses based on, for instance, the generalized Newton's method [27] (we discuss an experimental solver for it in Chapter 9). Finally, this additional flexibility does not necessarily need to come at a high cost. Our preliminary benchmarks show that the performance of Pre^* and $Post^*$ is quite competitive when compared to an established library such as WALi [48]. Furthermore, the performance of larger tools taking advantage of this approach should also be improved by the ability to avoid creating a copy of the program in the form of pushdown rules, as well as the ability to reuse the symbolic constraint system to perform various different analyses without recomputing the Pre^* and $Post^*$.

Chapter 9

Implementation of Newton's method

In this chapter we will describe an experimental solver based on Newton's method generalized for ω-continuous semirings [27, 28, 57]. One of the main advantages is that this technique makes it possible to compute the least fixed-points of non-linear equations whenever one is able to solve linear systems. For instance, it is possible to compute the least-fixed point of equation systems over, so called, counting abstractions, which do not necessarily satisfy the ascending chain condition but are commutative (i.e., the $\otimes$ operator is commutative). Note that this is quite different than techniques such as widening, which ensure the termination by over-approximating the least solution.

We will first recall some of the basic concepts behind the Newton's method in Section 9.1. Then, in Section 9.2, we describe the algorithms that we use for solving the equation systems. We focus on commutative semirings and also show that if they are additionally idempotent, the algorithms can be considerably simplified. Then in Section 9.3 we present an implementation of the counting semiring using semilinear sets. We discuss both the data structure and algorithms for the abstract domain as well as some of the challenges we encountered. Furthermore, we introduce some possible abstractions of semilinear sets that are significantly faster in practice. Finally, in Section 9.4, we present some benchmarks of our solver, again focusing mainly on semilinear sets and their abstractions. We conclude the chapter in Section 9.5.

Acknowledgement: This chapter is based on joint work with Maximilian Schlund during my research stay at the Technical University of Munich [71]. Furthermore, the described implementation is a joint project with Maximilian Schlund and Michael Kerscher from the Technical University of Munich.

9.1 Introduction and preliminaries

In this section we provide a brief introduction to the Newton's method, following [27] (which we also recommend for further details about the technique). We will also discuss the basic approaches to computing the Newton's sequence for the cases when the semiring is:

- not necessarily commutative or idempotent,
- commutative but not necessarily idempotent,
- both commutative and idempotent.

But first let us introduce some basic concepts that we will use throughout the chapter.

9.1.1 Semirings and polynomial equations

We have introduced the notion of a semiring in Definition 2.16 (Chapter 2). In this chapter, for clarity of the presentation, we will often write ab for $a \otimes b$. Recall that a semiring is ω-continuous if additionally:

- The relation $\sqsubseteq := \{(a, b) \in S \times S \mid \exists d \in S : a \oplus d = b\}$ is a partial order.
- Every ω-chain $(a_i)_{i\in\mathbb{N}}$ has a supremum wrt. $\sqsubseteq$ denoted as $\sup_{i\in\mathbb{N}} a_i$.
- Given arbitrary sequence $(b_i)_{i\in\mathbb{N}}$ and defining
$$\sum_{i\in\mathbb{N}} b_i := \sup\{b_0 \oplus \cdots \oplus b_i \mid i \in \mathbb{N}\}$$
for every sequence $(a_i)_{i\in\mathbb{N}}$, every $c \in S$ and every partition $(I_j)_{j\in J}$ of $\mathbb{N}$:
$$c\left(\sum_{i\in\mathbb{N}} a_i\right) = \sum_{i\in\mathbb{N}}(ca_i), \quad \left(\sum_{i\in\mathbb{N}} a_i\right)c = \sum_{i\in\mathbb{N}}(a_i c), \quad \sum_{j\in J}\left(\sum_{i\in I_j} a_i\right) = \sum_{i\in\mathbb{N}} a_i$$

In an ω-continuous semiring we define the Kleene-star $(-)^* : S \to S$ as

$$a^* = \sum_{k \in \mathbb{N}} a^k = \sup\{\bar{1} \oplus a \oplus a^2 \oplus \cdots \oplus a^k \mid k \in \mathbb{N}\} \text{ for } a \in S$$

In the following brief definitions, we use S to denote an ω-continuous semiring and $\mathcal{X}$ as a finite set of variables:

- An *monomial* is a finite expression of the form $a_1 X_1 \cdots X_k a_{k+1}$, where $k \in \mathbb{N}$, $a_1, \ldots, a_{k+1} \in S$ and $X_1, \ldots, X_k \in \mathcal{X}$.
- A *polynomial* is a finite sum of monomials.
- A *power series* is an expression of the form $\sum_{i \in I} m_i$, where I is a countable set and m_i is a monomial for every $i \in I$.

9.1.2 General case

From the point of view of an implementer, a useful way of thinking about the Newton's method is based on the idea of considering an equation system as a context-free grammar and the computation of Newton's sequence as an *unfolding* of that grammar [27, 58]. The i-th element of the Newton's sequence corresponds to the sum of yields of derivation trees of dimension up to i. We will use $X^{\langle i \rangle}$ and $X^{[i]}$ to denote a derivation tree of dimension exactly i and up to i respectively. This is best explained by an example — consider the following equation $X = aXXX + b$, its corresponding unfolding is as follows (with the base case being $X^{\langle 1 \rangle} = b$):

$$\begin{aligned} X^{\langle i \rangle} = {} & aX^{\langle i \rangle} X^{[i-1]} X^{[i-1]} \oplus aX^{[i-1]} X^{\langle i \rangle} X^{[i-1]} \oplus aX^{[i-1]} X^{[i-1]} X^{\langle i \rangle} \\ & \oplus aX^{\langle i-1 \rangle} X^{\langle i-1 \rangle} X^{[i-2]} \oplus aX^{\langle i-1 \rangle} X^{[i-2]} X^{\langle i-1 \rangle} \oplus aX^{[i-2]} X^{\langle i-1 \rangle} X^{\langle i-1 \rangle} \\ & \oplus aX^{\langle i-1 \rangle} X^{\langle i-1 \rangle} X^{\langle i-1 \rangle} \end{aligned}$$

and we also have that $X^{[i]} = X^{[i-1]} \oplus X^{\langle i \rangle}$. First note that the unfolding gives rise to a linear equation system, so every iteration of Newton's method requires solving a such a system. Using the idea of grammar unfolding immediately gives us a basic formulation of a recursive algorithm Newton's sequence, provided that we are able to solve a linear equation system.

9.1.3 Commutative semirings

As observed in [27] when a semiring is commutative, the approach to computing the Newton's sequence can be expressed more succinctly. The observation made in the paper is that the least solution for: $D\mathbf{f}|_{\mathbf{u}}(\mathbf{X}) \oplus \mathbf{v} = \mathbf{X}$ (where $D\mathbf{f}|_{\mathbf{u}}$ stands for the differential of $\mathbf{f}$ at $\mathbf{u}$) is equal to $D\mathbf{f}|^*_{\mathbf{u}}(\mathbf{v}) = \frac{\partial \mathbf{f}}{\partial \mathcal{X}}\Big|^*_{\mathbf{u}} \mathbf{v}$ where $\frac{\partial \mathbf{f}}{\partial \mathcal{X}}\Big|_{\mathbf{u}}$ is the Jacobian of the vector of the power series $\mathbf{f}$ evaluated at $\mathbf{u}$. This allows simplifying the Newton sequence to:

$$\mathbf{v}^{(0)} = \mathbf{f}(\bar{\mathbf{0}}) \qquad \mathbf{v}^{(i+1)} = \mathbf{v}^{(i)} \oplus D\mathbf{f}|^*_{v^{(i)}}(\boldsymbol{\delta}^{(i)}) = \mathbf{v}^{(i)} \oplus \left.\frac{\partial \mathbf{f}}{\partial \mathcal{X}}\right|^*_{\mathbf{v}^{(i)}} \boldsymbol{\delta}^{(i)}$$

Note that we take the Kleene star of the Jacobian, which together with multiplying it by $\boldsymbol{\delta}^{(i)}$ corresponds to solving the linear equation system. This is based on the idea that in the commutative case, the grammar unfolding can be seen as a right-linear equation system. Thus the example from the previous subsection can be seen as:

$$X^{\langle i \rangle} = \underbrace{3aX^{[i-1]}X^{[i-1]}}_{a} X^{\langle i \rangle} \oplus \underbrace{3aX^{[i-2]}X^{\langle i-1 \rangle}X^{\langle i-1 \rangle} \oplus aX^{\langle i-1 \rangle}X^{\langle i-1 \rangle}X^{\langle i-1 \rangle}}_{b}$$

And as known from Arden's Lemma the least solution for an equation system $X = aX \oplus b$ is a^*b. Note that this directly corresponds to the Newton's sequence presented above, where we take the Kleene star of the Jacobian matrix and multiply it by $\boldsymbol{\delta}^{(i)}$. This correspondence to grammar unfolding also provides us with means to compute the $\boldsymbol{\delta}^{(i)}$.

9.1.4 Commutative and idempotent semirings

When a semiring is not only commutative but also idempotent the Newton iteration becomes far more straightforward. Again, following [27], the computation of a Newton sequence can be accomplished using the following formula:

$$\mathbf{v}^{(i+1)} = D\mathbf{f}|^*_{\mathbf{v}^{(i)}}(\mathbf{f}(\bar{\mathbf{0}})) = \left.\frac{\partial \mathbf{f}}{\partial \mathcal{X}}\right|^*_{\mathbf{v}^{(i)}} \mathbf{f}(\bar{\mathbf{0}})$$

which immediately gives us a simple iterative method to compute the Newton sequence.

9.2 Algorithms

In this section we will discuss how the Newton iteration can be implemented and what is the complexity of the algorithms. As already mentioned, we will break down the computation into two main parts — the computation of δ and solving the linear equation system arising at each Newton iteration. Thus, the general structure of the algorithm remains the same and we only change the linear solver and the delta generator in order to handle the cases of different kinds of semirings. After presenting the generic algorithm, we will focus on the commutative cases and in a subsequent section on an interesting commutative abstraction — semilinear sets. We will start with discussing the generic algorithm and then go on to describe the linear solvers and δ computations for the two cases.

9.2.1 Generic algorithm for Newton's sequence

input : number of iterations N, column vector $\mathbf{F}$ of polynomials, column vector of variables **variables**
output: column vector of semiring values **values**

```
InitLinearSolver(F, variables);
delta ← InitDeltaGenerator(F, variables ) ;        /* Computes δ^(1) */
values ← Map (λx → 0̄) variables ;                  /* Computes v^(0) */
for i ∈ 1 ... N do
    update ← SolveLinear(values, delta ) ;         /* Computes u^(i) */
    delta ← GenerateDelta(values, update ) ;       /* Computes δ^(i+1) */
    values ← UpdateNewton(values, update ) ;       /* Computes v^(i) */
end
return values
```

Algorithm 1: Generic algorithm for computing the Newton sequence.

The structure of the generic algorithm (Algorithm 1) is quite simple — it first initializes the linear solver (`InitLinearSolver()`) and what we call the delta generator (`InitDeltaGenerator()`), then it performs a given number of Newton iterations. At each of them it solves a linear equation system (`SolveLinear()`), computes a new $\boldsymbol{\delta}$ (`ComputeDelta()`) and updates the current Newton approximation (`UpdateNewton()`).

9.2.2 Commutative cases

There is a lot of similarity in our approach to the two cases when the semiring is commutative, thus this section applies to both of them.

As mentioned in the preliminaries, at each of the Newton steps one needs to solve a system of linear equations. If the semiring satisfies the ascending chain condition, this can be done using standard Kleene iteration. However, many interesting domains do not satisfy this requirement (e.g., semilinear sets described in Section 9.3). In those cases we compute the Kleene star of the Jacobian matrix of the polynomials, which when multiplied by $\boldsymbol{\delta}^{(i)}$ (for the current i) constitutes the solution to the linear equation system.

We have decided to implement the Kleene star computation in a symbolic way, i.e., substitute a fresh variable for each constant and run the algorithm on the resulting matrix (this can be viewed as computation over the free semiring). This allows us to avoid computing it in every iteration — we simply substitute the values for variables and evaluate the resulting expressions (remembering about the constants corresponding to the fresh variables). The main advantage of this approach is not only the ability to avoid recomputing the star of Jacobian matrix, but it also opens a possibility of detecting common subexpressions and avoiding recomputing them. This can be significant for domains with expensive semiring operations.

However, computing the Kleene star over the free semiring is exponential in the amount of space used[1]. We solve the problem using hash-consing [31] (one can also think about it in terms of value numbering [16] as used in compilers or the main idea behind Reduced Ordered BDDs [12]). Our data structure for the free semiring is implemented as a expression tree (i.e., an AST) with the additional invariant that we never create two identical nodes. This allows detecting when some expressions have identical values simply by pointer equality and at the same time reduces the memory usage since we can share the subexpressions (and avoid copying them that causes the exponential behavior).[2]

We have implemented the Kleene star computation using three different algorithms — the generalized Floyd-Warshall algorithm and two recursive divide-and-conquer ones. All of them are $\Theta(n^3)$; we discuss one of the recursive algorithms and its running time in Appendix D.1. It is interesting to note that the semiring expressions that arise in each of the algorithms are different, which can have a significant impact on the performance of the solver for certain semirings.

[1] There is a linear number of updates of each cell, but at each such update we can more than double the size of the expressions involved

[2] Obviously for this to work we require that the nodes in the AST are immutable.

9.2.3 Non-idempotent and commutative case

In this section we will discuss the implementation of the linear solver and delta generator for commutative (but not necessarily idempotent) semirings. When discussing the linear solver we will focus on the variant using the Kleene star computation. As already mentioned, it is also possible to use other techniques such as Kleene iteration, but that requires additional properties such as the ascending chain condition.

Since we have decided to compute a symbolic Kleene star of the Jacobian matrix in order to detect some of the common subexpressions, the computation is performed during the initialization of the linear solver and on subsequent calls to `SolveLinear()` we simply evaluate the symbolic matrix using the supplied values.

input : column vector $\mathbf{F}$ of polynomials, column vector **variables**
output: compute and store $\mathbf{J}^*$ and map^R and **variables** for subsequent use

$(\mathbf{J}^F, \mathsf{map}^R) \leftarrow$ `MakeFree(ComputeJacobian(`$\mathbf{F}$`))` ;
$\mathbf{J}^* \leftarrow$ `ComputeStar(`$\mathbf{J}^F$`)` ;

Procedure InitLinearSolver

input : current values of variables **values** and current **delta**
output: Newton **update**

$\mathsf{map} \leftarrow$ `UpdateMap(MakeMap(`**variables**, **values**`)`, map^R `)` ;
return `Evaluate(`$\mathbf{J}^*$, map `)` $\times$ **delta**

Procedure SolveLinear

Following the results from [27] that were briefly presented in Section 9.1, the procedure for updating the current Newton approximation is simply the $\oplus$ of **values** and **update** vectors.

input : current **values** and **update**
output: new **values**

return values $\oplus$ **update**

Procedure UpdateNewton

The initialization of the delta generator (Procedure InitDeltaGenerator) follows directly from [27] (again presented in Section 9.1).

input : column vectors: **F** with polynomials, **variables**
output: return the initial value of delta

return `Evaluate(`$\mathbf{F}, \mathbf{variables}, \bar{\mathbf{0}}$`)`

Procedure InitDeltaGenerator

input : column vectors: **values** with the current values and **update** with the current Newton update
output: column vector of semiring values corresponding to the new δ

```
foreach polynomial F at index i in F do
    foreach subset S of the multiset of variables where each variable v appears
    at most Degree(F, v) times do
        delta[i] ← 0̄;
        foreach monomial M in F do
            value_M ← Coefficient(M );
            foreach variable v in M do
                s ← Multiplicity(S, v);
                m ← Multiplicity(M v);
                if s < m then
                    value_M ← value_M ⊗ (m choose s) ⊗ (Lookup(values_map, v ))^(m−s)
                    ⊗ (Lookup(update_map, v ))^s;
                else
                    value_M ← 0̄;
                    continue with the next monomial
                end
            end
            delta[i] ← delta[i] ⊕ value_M;
        end
    end
end
return delta
```

Procedure GenerateDelta

The computation of δ (Algorithm GenerateDelta) is not entirely trivial — for each polynomial we consider all possible ways of selecting two or more variables to be evaluated at the last Newton update (with the rest being evaluated at the current Newton values). One way of looking at the problem is to consider a multiset of variables that appear in a monomial M. We want to create all possible k-multicombinations S where $2 \leq k$ and where k is smaller or equal to the degree of the monomial. Let us denote by S_v the multiplicity of variable v in S and define M_v similarly. Then for each variable v, if its M_v is greater than or equal to S_v we can say that there are $\binom{M_v}{S_v}$ unfoldings with exactly S_v instances of v evaluated at the last Newton update. Since we need to examine each monomial, then find the multicombinations and examine each variable, the complexity of the procedure is:

$$\mathcal{O}\left(|\mathcal{M}| * |\mathcal{X}| * \sum_{k=2}^{D} \binom{|\mathcal{X}| + k - 1}{k}\right)$$

where $\mathcal{M}$ is a set of all monomials, $\mathcal{X}$ is the set of all variables and D is a highest degree of a monomial. As an example, for a quadratic equation system[3] (i.e., $D = 2$), the above simplifies to $\mathcal{O}(|\mathcal{M}| * |\mathcal{X}|^3)$.

Finally, putting everything together we have that computing N iterations of the above procedure is (assuming that we use symbolic Kleene star computation):

$$\mathcal{O}\left(N\left(|\mathcal{X}|^3 + |\mathcal{M}| * |\mathcal{X}| * \sum_{k=2}^{D} \binom{|\mathcal{X}| + k - 1}{k}\right)\right)$$

9.2.4 Idempotent and commutative case

In this section we discuss the idempotent case — it is not only the simplest one but also has the lowest complexity. We have N iterations and at each we need to solve the linear system of equations, in the general case this corresponds to evaluating a matrix constructed by the Kleene star computation. As already discussed the cost of computing and evaluating the matrix is cubic in the number of variables. Thus the total complexity of the procedure is simply $\mathcal{O}(N * |\mathcal{X}|^3)$.

The procedures `InitLinearSolver()` and `SolveLinear()` are the same whenever the semiring is commutative (i.e., no matter whether it is also idempotent or not). However, the `InitDeltaGenerator()` and `GenerateDelta()` can be

[3] Such systems arise when using weighted [66, 67] or communicating [10, 11] pushdown systems.

significantly simplified. Since $\boldsymbol{\delta}$ for this case does not differ between iterations, we pre-compute it in `InitDeltaGenerator()` and then reuse it at each `GenerateDelta()`.

input : column vectors with n elements: $\mathbf{F}$ with polynomials, **variables** with variables
output: compute and store idelta for subsequent use

idelta $\leftarrow$ `Evaluate(`$\mathbf{F}, \mathbf{variables}, \bar{\mathbf{0}}$`)` ;
return idelta

Procedure InitDeltaGenerator

input : column vectors with n elements: **values** with the current values and **update** with the current Newton update
output: the already computed idelta

return idelta

Procedure GenerateDelta

input : current **values** and **update**
output: new **values**

return update

Procedure UpdateNewton

9.3 Semilinear sets and their abstractions

9.3.1 Definition

Let us start with some preliminary definitions of linear and semilinear sets [33]. First of all, we say that a nonempty set $L \subseteq \mathbb{N}^n$ is linear if it can be expressed as:

$$\mathbf{v}_0 + \mathbb{N}\mathbf{v}_1 + \cdots + \mathbb{N}\mathbf{v}_m$$

where $\mathbf{v}_0, \ldots, \mathbf{v}_m \in \mathbb{N}^n$ for some $m \in \mathbb{N}$. We will say that $\mathbf{v}_0$ is the offset and $\mathbf{v}_1, \ldots, \mathbf{v}_m$ are the generators. Furthermore, we will often denote such sets as a pair $(\mathbf{v}, G)$ consisting of the offset $\mathbf{v}$ and set of generators $G \subseteq \mathbb{N}^n$. We can define an addition operator $\oplus_L$ for linear sets:

$$(\mathbf{v}_1, G_1) \oplus_L (\mathbf{v}_2, G_2) = (\mathbf{v}_1 + \mathbf{v}_2, G_1 \cup G_2)$$

A semilinear set is a finite union of linear sets. Let us also define the interpretation of this representation of semilinear sets, i.e., the (potentially infinite) set of actual vectors corresponding to it.

$$\mathcal{I}[\![S]\!] = \{\mathbf{v} + k_1\mathbf{g}_1 + \cdots + k_n\mathbf{g}_n \mid n \in \mathbb{N},\ (\mathbf{v}, G) \in S, \\ k_1, \ldots, k_n \in \mathbb{N},\ \mathbf{g}_1, \ldots, \mathbf{g}_n \in G\}$$

Having that, we will also define an ordering on vector representation of semilinear sets (it will be useful later on):

$$S_1 \sqsubseteq S_2 \iff \mathcal{I}[\![S_1]\!] \subseteq \mathcal{I}[\![S_2]\!]$$

Semilinear sets also form a semiring $(\mathcal{S}, \oplus_S, \otimes_S, \bar{0}_S, \bar{1}_S)$ in the following way:

- Operator $\oplus_S$ is simply a union of the semilinear sets $S_1 \oplus_S S_2 = S_1 \cup S_2$ and thus can be implemented in $\mathcal{O}(|S_1| + |S_2|)$.
- Operator $\otimes_S$ corresponds to adding all pairs of linear sets

  $$S_1 \otimes_S S_2 = \{L_1 \oplus_L L_2 \mid L_1 \in S_1,\ L_2 \in S_2\}$$

  which is quite expensive — $\mathcal{O}(|S_1||S_2|)$.
- the $\bar{0}_S$ element is simply the empty set, $\bar{0}_S = \emptyset$
- the $\bar{1}_S$ element is a singleton set where the only element is a linear set with the offset equal to $\mathbf{0}$ and no generators, i.e., $\bar{1}_S = \{(\mathbf{0}, \emptyset)\}$

It is quite easy to see that all the semiring properties are satisfied. More importantly, it should be clear that the $\otimes_S$ operator is commutative (since $\oplus_L$ is). The only remaining thing is the definition of Kleene star $(-)^* : \mathcal{S} \to \mathcal{S}$ operator. In order to define it, we first introduce a similar operator for linear sets, $(-)^* : \mathcal{L} \to \mathcal{S}$:

$$(\mathbf{v}, G)^* = \bar{1}_S \oplus_S (\mathbf{v}, \{\mathbf{v}\} \cup G)$$

With that we can inductively define the star operator for semilinear sets in the following way:

$$S^* = \begin{cases} \bar{1}_S & \text{if } S = \emptyset \\ L^* \otimes_S (S \setminus \{L\})^* & \text{otherwise; where } L \in S \end{cases}$$

The intuition behind this is that for every linear set we can either include it any number of times (including 0, thus the $\bar{1}_S$) and then multiply with the star of the remaining linear sets. Note that this definition relies on the fact that

$\otimes_S$ is commutative. Finally, it should be clear that this is the most expensive operation, in fact it is exponential — in the worst case, at each step of the recursion we double the number of linear sets.

It should be clear that the domain does not satisfy the ascending chain condition and thus using Kleene iteration (in general) would not allow us to reach the least fixed point of a system of equations in a finite number of steps. However, we can compute it using the algorithms presented in the previous section.

9.3.2 Challenges

The above representation of semilinear sets can be directly implemented. However, due to the complexity of the $\otimes_S$ and $(-)^*$ operations making it practical is much more challenging. One of the optimizations that we have implemented is that we share common vectors and linear sets that appear in different linear sets and semilinear sets respectively (using the idea of hash-consing again). But the main challenge is the space complexity of $\otimes_S$ and $(-)^*$, e.g., $\otimes$ is used extensively and $S_1 \otimes_S S_2$ has space complexity $\mathcal{O}(|S_1||S_2|)$. As already mentioned, an important part of computing the Newton's approximations is the evaluation of a Kleene star of Jacobian matrix. This corresponds (in the worst case) to computing a symbolic expression of height equal to the number of variables with $\otimes_S$ and $(-)^*$ at each level. Unfortunately this results in an exponential space complexity.

Fortunately, during our experiments we have observed that most of the generated vectors and linear sets are not necessary, i.e., they duplicate already existing information. Consider, for example, the following linear set:

$$(\langle 0,0\rangle, \{\langle 1,1\rangle, \langle 2,2\rangle\})$$

Clearly the first generator makes the second one redundant. Of course, we can also have a situation where a few different generators make some other one unnecessary. Similarly we can have redundancy between linear sets themselves. This leads us to experiment with various ways of "simplifying" away generators or linear sets that we can prove are unnecessary. For a generator $\mathbf{g}$ we check whether there exists a multiset consisting of other generators that sum to $\mathbf{g}$. If that is the case, then we can safely remove $\mathbf{g}$. We use similar ideas with respect to linear sets. Unfortunately, this approach amounts to solving a problem similar to the, so called, subset-sum problem which itself is a variant of knapsack problem [17]. It is known to be NP-complete, however, using memoization we are able to reuse many already computed results and the whole procedure is in most cases fast enough to enable us to solve some equation systems that we

would not be able to solve without the simplification (due to the blowup in the size of linear and semilinear sets). We will discuss this topic a bit more in the Section 9.4.

9.3.3 Abstractions

We have also developed two ways of improving the performance of semilinear sets at the cost of their precision. In other words, we have developed some abstractions of semilinear sets.

The first one is based on the idea of dividing every generator of every linear set by the greatest common divisor of its elements. Let us consider an example — if we have a linear set $(\langle 0,0\rangle, \{\langle 2,2\rangle\})$ then we can consider another linear set $(\langle 0,0\rangle, \{\langle 1,1\rangle\})$ to be its over-approximation. It will include all the vectors of the first one (e.g., $\langle 2,2\rangle$, $\langle 4,4\rangle$, etc.), but it will also contain $\langle 1,1\rangle$ or $\langle 3,3\rangle$. At this point it might seem that we have not improved anything, but when combined with the idea of simplification from the previous section, it often allows for more compact representation. For instance, the linear set $(\langle 0,0\rangle, \{\langle 2,2\rangle, \langle 5,5\rangle\})$ would be abstracted to $(\langle 0,0\rangle, \{\langle 1,1\rangle\})$. It should be quite easy to see that this approach provides a safe over-approximation. For additional insight one might consider the graphic representation of the two-dimensional case — a generator gives rise to a set of points laying on some line. Our abstraction corresponds to including all the points on that line, i.e., "forgetting" what points are not included.

The second abstraction is more complex and focused around the idea of "collapsing" a set of linear sets (i.e., a semilinear set) into a pair of two sets — one of offsets and the other one of generators. We will call this structure a *multilinear set*. The intuition behind it is that we can choose any of the offsets and then use the generators as in the case of linear sets. As expected multilinear sets form a semiring $(\mathcal{M}, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$, where:

- The set $\mathcal{M}$ is defined as $\mathcal{M} = \{(V,G) \mid V \subseteq \mathbb{N}^n,\ G \subseteq \mathbb{N}^n,\ V \neq \emptyset\} \cup \{\bot\}$.
- The $\oplus_M$ operator is defined as:

  $$(V_1, G_1) \oplus_M (V_2, G_2) = (V_1 \cup V_2, G_1 \cup G_2)$$

  and additionally (if one of the arguments is $\bot$) for all $M \in \mathcal{M}$ we have $M \oplus_M \bot = \bot \oplus_M M = M$. The complexity is $\mathcal{O}(|V_1| + |V_2| + |G_1| + |G_2|)$.
- The $\otimes_M$ operator is defined as:

  $$(V_1, G_1) \otimes_M (V_2, G_2) = (\{v_1 + v_2 \mid v_1 \in V_1,\ v_2 \in V_2\}, G_1 \cup G_2)$$

and additionally (if one of the arguments is $\perp$) for all $M \in \mathcal{M}$ we have $M \otimes_M \perp = \perp \otimes_M M = \perp$. The complexity of this computation is $\mathcal{O}(|V_1||V_2| + |G_1| + |G_2|)$.

- The $\bar{0}_M$ element is $\perp$.
- The $\bar{1}_M$ element is $(\{\mathbf{0}\}, \emptyset)$.

Note that we require that the set of offsets is never empty — an empty set of offsets would correspond to an empty interpretation, no matter what are the contents of the set of generators, i.e., we would have an infinite number of elements with an empty interpretation. Thus we disallow empty set of offsets and introduce a special $\perp$ element that corresponds to empty interpretation. The definition of the star operator is not difficult and is quite different compared to the semilinear sets:

$$(V, G)^* = (\{\mathbf{0}\}, V \cup G)$$

Its complexity is $\mathcal{O}(|V| + |G|)$, which is much faster than what we can achieve with the semilinear sets and, we conjecture, is one of the big factors why this abstraction performs much better in practice.

Of course, we can also define the interpretation of a multilinear set:

$$\mathcal{I}[\![(V, G)]\!] = \{\mathbf{v} + k_1\mathbf{g}_1 + \cdots + k_n\mathbf{g}_n \mid n \in \mathbb{N},\ \mathbf{v} \in V,\\ k_1, \ldots, k_n \in \mathbb{N},\ \mathbf{g}_1, \ldots, \mathbf{g}_n \in G\}$$

The definition of ordering for multilinear sets is analogous to the one for semilinear sets.

Having the interpretation and ordering, we are able to clarify the relationship between semilinear and multlinear by establishing a Galois connection between them. In the case that neither argument is $\bar{0}$:

$$\alpha(S) = \Big(\{\mathbf{v} \mid (\mathbf{v}, G) \in S\},\ \bigcup\{G \mid (\mathbf{v}, G) \in S\}\Big)$$
$$\gamma((V, G)) = \{(\mathbf{v}, G) \mid \mathbf{v} \in V\}$$

and otherwise: $\alpha(\emptyset) = \perp$ and $\gamma(\perp) = \emptyset$. As we will see below, this actually defines a Galois insertion (however, this property is not essential for our purposes).

Lemma 9.1 *The tuple* $(\mathcal{S}, \alpha, \gamma, \mathcal{M})$ *forms a Galois insertion.*

Proof. It is easy to see that α and γ are monotone, so we need to only prove that $\alpha \circ \gamma = \lambda x.x$ and $\lambda x.x \sqsubseteq \gamma \circ \alpha$. Since the cases when arguments are $\emptyset$ or $\perp$ are simple, below we consider the other possibilities.

Let us prove the first claim:

$$
\begin{aligned}
&(\alpha \circ \gamma)(V, G) \\
&\quad = [\text{definition of } \gamma] \\
&\alpha\left(\{(\mathbf{v}, G) \mid \mathbf{v} \in V\}\right) \\
&\quad = [\text{definition of } \alpha] \\
&\left(\{\mathbf{v'} \mid (\mathbf{v'}, G') \in \{(\mathbf{v}, G) \mid \mathbf{v} \in V\}\},\ \bigcup\{G' \mid (\mathbf{v'}, G') \in \{(\mathbf{v}, G) \mid \mathbf{v} \in V\}\}\right) \\
&\quad = [\text{simplification}] \\
&(V, G)
\end{aligned}
$$

The second case is only slightly more involved.

$$
\begin{aligned}
&(\gamma \circ \alpha)S \\
&\quad = [\text{definition of } \alpha] \\
&\gamma\left(\{\mathbf{v} \mid (\mathbf{v}, G) \in S\},\ \bigcup\{G \mid (\mathbf{v}, G) \in S\}\right) \\
&\quad = [\text{definition of } \gamma] \\
&\left\{(\mathbf{v}, \bigcup\{G \mid (\mathbf{v'}, G) \in S\}) \mid \mathbf{v} \in \{\mathbf{v''} \mid (\mathbf{v''}, G) \in S\}\right\} \\
&\quad = [\text{simplification}] \\
&\left\{(\mathbf{v}, \bigcup\{G \mid (\mathbf{v'}, G) \in S\}) \mid (\mathbf{v}, G') \in S\right\}
\end{aligned}
$$

At this point it is not difficult to see that for every linear set $(\mathbf{v}, G) \in S$ we will have a linear $(\mathbf{v'}, G') \in (\alpha \circ \gamma)S$ such that $\mathbf{v} = \mathbf{v'}$ and $G \subseteq G'$. In other words, for every vector that can be generated from S, it will also be possible to generate it using the $(\gamma \circ \alpha)S$. Thus $S \sqsubseteq (\gamma \circ \alpha)S$, i.e., $\lambda x.x \sqsubseteq \gamma \circ \alpha$. $\square$

9.4 Examples and experiments

In this section we will present some simple but interesting examples of how the domain can be useful for program analysis. We will briefly discuss two potential areas of application of semilinear sets.

One of possible applications of semilinear sets is to count the number of communication/synchronization actions (as suggested in [10, 11]) over each channel. Consider the following program:

```
proc main
  a?(x);
  if
    :: x = 0     => skip
    :: !(x = 0) => call main
  fi;
  b!()
end
```

and let us focus only on the channels used. The corresponding language is $\{a^k b^k \mid k \in \mathbb{N}, 0 < k\}$. However, using Kleene iteration we would not be able to reach the least solution of the corresponding equation system due to the fact that we could always add a new longer word. Running our tool on such a system (generated with the $Post^*$ using our pushdown library presented in Chapter 8), the least solution to describing the point when the program terminates is:

```
{ <[(a, 1)(b, 1)] : >,
  <[(a, 2)(b, 2)] : [(a, 1)(b, 1)]> }
```

which corresponds to two linear sets — one representing the vector $\langle 1, 1\rangle$ and the second describing the set $\{a^k b^k \mid k \in \mathbb{N}, 2 \leq k\}$. Note that the set could also be represented using just one linear set $(\langle 1, 1\rangle, \{\langle 1, 1\rangle\})$, which corresponds to the same set of vectors. Currently we do not perform this kind of simplification, but it might be an interesting idea for future work.

Another potential application of this domain is counting the number of times some resource is acquired and released. For instance, consider a recursive program performing locking and unlocking of a reentrant lock:

```
proc foo
  Lock!();
  if
    :: true => call foo
    :: true => skip
  fi;
  call bar
end

proc bar
  if
    :: true => Unlock!()
    :: true => call bar
  fi
```

```
end

proc main
  call foo
end
```

The number of locks and unlocks is the same (even though it might be arbitrarily large) and our analysis is capable of proving that. The semilinear set corresponding to the point when the program terminates is:

```
{ <[(Lock, 1)(Unlock, 1)] : [(Lock, 1)(Unlock, 1)]> }
```

Of course, since the domain is commutative it does not differentiate between different locking orders. Thus, it is not possible to detect data races in this situation.

Finally, it is also possible to use semilinear sets for points-to analysis as discussed in [27].

Now we will briefly discuss the performance of our abstract domain. For that we will run a few benchmarks based on the $Post^*$ constraints of some simple programs and compare the solving time for semilinear sets (with and without simplification) as well as its abstractions. The example programs used in the tests are available in the Appendix D.2. We present the results of solving the equation systems in Table 9.1.

One of the things that should be clear from those results is that the simplification step is essential. Even though its asymptotic complexity is very high, it often allows to avoid the blowup in the size of semilinear and linear sets. We can also conclude from those results that most of the elements of semilinear sets (and linear sets) that arise during the computation are in fact redundant and can be removed. Furthermore, as one could expect, the abstraction to multilinear sets is much faster in practice. Finally, abstraction based on a "divider" (where we divide the elements of the vectors by their GCD) does not always improve the performance and in some cases it actually makes it noticeably worse. We believe that this is due to the fact that in some cases the vectors do not have a GCD other than 1, which means that all the effort of trying to find one is wasted and the divider has no effect (this is the case in examples 1 and 3). On the other hand, example 2 is similar to the first one but it does create vectors that can be abstracted using a divider and consequently the solving time is reduced considerably. The same effect can be observed in case of multilinear sets, but on a smaller scale.

	Example 1	Example 2	Example 3
Semilinear sets without simplification	$>$ 60 s and 1GB	$>$ 60 s and 1GB	$>$ 60 s and 1GB
Semilinear sets with simplification	1519 ms	1516 ms	3330 ms
Multilinear sets without simplification	14 ms	32 ms	$>$ 60 s and 1GB
Multilinear sets with simplification	14 ms	36 ms	11 ms
Semilinear sets with simplification and divider	2325 ms	128 ms	6877 ms
Multilinear sets with simplification and divider	14 ms	27 ms	19 ms

Table 9.1: Time to solve constraints for different domains and examples.

9.5 Conclusions

In this chapter we have discussed an experimental implementation of the Newton's method [57, 27]. It allows to solve non-linear systems of equations, provided that one is able to solve the linear ones. Since interprocedural analysis is in general a source of non-linear equation systems, this is an important advantage in the context of this thesis. In case of commutative abstractions, such as semilinear sets, the Newton's method provides a way of computing the least fixed point solution whereas the usual Kleene iteration would not reach it in a finite number of steps. This is also the reason for our focus on semilinear sets. Unfortunately they are very expensive in terms of computational complexity. However, we managed to limit the space blowup with some simplification techniques, as well as by developing abstractions, which trade some of the precision for performance.

We believe our results are quite encouraging and show that it is already possible to perform interesting analyses on smaller programs. At the same time there is still a lot of potential for improvements and future work in terms of the solver itself, new abstract domains as well as the semilinear sets, their abstractions and applications. Finally, it is interesting to note how well the Newton's method fits into existing techniques such as weighted and communicating pushdown systems, allowing us to use new abstract domains in existing approaches and tools.

Chapter 10

Conclusions

We believe that automated or semi-automated analysis techniques will be increasingly important in building modern software. From bug finding tools to verification tools capable of proving lack of certain classes of errors, their main building block will probably be some combination of static analysis, model checking and abstract interpretation. Furthermore, those areas will most likely see even more research activity in the next decades.

We have explored the algebraic approach to static analysis and model checking and reformulated some of the concepts of abstract interpretation in this setting in Chapter 3. It turns out that expressing many analyses in terms of flow algebras or semirings fits quite well into the abstract interpretation framework. Moreover, we introduced and proved some basic properties and concepts that later turned out to be also useful in the setting of communicating pushdown systems (e.g., over-approximation and inducing flow algebras).

Apart from that, we have looked at the communicating and weighted pushdown systems, first contrasting them in Chapter 4 and then investigating the requirements they impose on the used algebraic structure. We have shown in Chapter 5 that some of the requirements imposed by the original formulations using semirings can be in fact relaxed. By introducing a new variant of Pre^* and $Post^*$ algorithms we have shown that it is possible to use flow algebras in place of semirings, which corresponds more closely to many classical analyses.

We believe that this contributes both to a better understanding of weighted pushdown systems as well as bringing them closer to the monotone frameworks and providing some additional flexibility for designers of abstract domains.

In Chapter 6 we revisited an analysis problem encountered in [36] where the addition of aspect-oriented features to a process calculus resulted in a potential recursive structure of the processes. The original work disallowed this situation, but we have shown how to analyze such a process calculus when this restriction is lifted. One of the main ideas behind our approach is to use the stack of the pushdown system to represent a process itself. Thus the Pre^* and $Post^*$ computations correspond to all possible shapes of a process in the future. Using communicating pushdown systems in this way allowed us to reason about reachability of configurations in a concurrent setting. Furthermore, the technique is also useful in a sequential setting when using the weighted pushdown systems.

The analysis of the aspect-oriented process calculus motivated us to improve the ith-prefix and i^{th}-suffix abstractions of [14]. From the theoretical perspective we developed an approach of encoding symbolic constraints in the prefix/suffix language that allows handling of message passing in a more efficient manner. At the same time we described an compact data structure to represent such languages, i.e., Reduced DFA. We additionally provided efficient algorithms for the $\oplus$ and $\otimes$ operations and performed a preliminary evaluation of our implementation.

In some of our work we have relied on a well-known Weighted Automata Library (WALi) [48]. However, some of its design choices made it difficult to experiment with techniques such as the ones based on Newton's method [27]. Therefore, we have decided to create a library that would provide some more flexibility in this regard and described it in Chapter 8. Using this opportunity we have also developed a slightly different approach to the Pre^* and $Post^*$ algorithms. Instead of considering the pushdown system as a set of pushdown rules, we have created algorithms that can work on, what we call, pushdown graphs. We believe that this is an important practical contribution, since the representation is very close to the program graphs and thus allows to use them directly, i.e., without converting the program to pushdown rules. Furthermore, the library provides additional benefits of thread-safety as well as the possibility of using custom constraint solvers.

Finally, in Chapter 9 we described an experimental equation solver based on Newton's method [27]. It enables computing the least fixed-points of non-linear equation systems over ω-continuous semirings, whenever one is able to solve such linear systems. This has some exciting consequences, e.g., we are able to compute least fixed-points of equation systems for abstract domains such as semilinear sets. Despite high complexity of this domain, our initial experiments are quite encouraging. Moreover, we developed abstractions of semilinear sets

that are much faster in practice.

We believe that this thesis contributes to the efforts of not only improving the techniques from static analysis, model checking and abstract interpretation but also to making them easier to use in practice. Our work is quite diverse and on one hand includes quite theoretical developments in Chapters 3 where we introduced flow algebras and explored their use in the abstract interpretation framework. Furthermore, in Chapter 5 we considered their application to weighted and communicating pushdown systems, thus showing that some of the previous requirements of WPDS and CPDS are not necessary. Moreover, we presented a novel application of communicating pushdown systems to an aspect-oriented process calculus in Chapter 6. Not only that, but we have also contributed to the development of better implementations by describing new algorithms and data structures. In Chapter 7 we introduced the concept of RDFA for efficient representation of finite languages with symbolic encoding of constraints. And in Chapter 8 we presented graph-based algorithms for the variants of Pre^* and $Post^*$ algorithms described in Chapter 5. Finally, we also developed an experimental solver based on Newton's method in Chapter 9, which can be applied to the constraint systems arising in Pre^* and $Post^*$ computations. As such it constitutes an interesting advantage of using an algebraic approach to static analysis.

Finally, we think that there is still a lot of room for future work. One of the main potential projects that would be worth pursuing is the development of a larger tool based on CPDS, flow algebras and Newton's method that is capable of dealing with the complexity of mainstream programming languages. We believe the thesis does provide good foundations for that work, yet we also expect that implementing such a tool would lead to many new and interesting challenges.

Appendix A

Proofs for Chapter 3

A.1 Proof of Lemma 3.7

For two flow algebras $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$, $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ and a Galois insertion (L, α, γ, M), the following are equivalent:

1. $\bar{1}_L = \gamma(\bar{1}_M)$
2. $\alpha(\bar{1}_L) = \bar{1}_M$ and $\bar{1}_L \in \gamma(M)$

Proof. We will show that the above statements imply each other.

1. First we show that the first one implies the second one

$$\begin{aligned} \bar{1}_L &= \gamma(\bar{1}_M) \\ \alpha(\bar{1}_L) &= \alpha(\gamma(\bar{1}_M)) \\ \alpha(\bar{1}_L) &= \bar{1}_M \end{aligned}$$

2. And now in the opposite direction

$$\begin{aligned}
\alpha(\bar{1}_L) &= \bar{1}_M \\
\alpha(\gamma(m')) &= \bar{1}_M \quad \text{for some } m' \\
m' &= \bar{1}_M \\
\gamma(m') &= \gamma(\bar{1}_M) \\
\bar{1}_L &= \gamma(\bar{1}_M)
\end{aligned}$$

□

A.2 Proof of Lemma 3.8

For two flow algebras $(L, \oplus_L, \otimes_L, \bar{0}_L, \bar{1}_L)$, $(M, \oplus_M, \otimes_M, \bar{0}_M, \bar{1}_M)$ and a Galois insertion (L, α, γ, M), the following are equivalent:

1. $\forall m_1, m_2 : \gamma(m_1) \otimes_L \gamma(m_2) = \gamma(m_1 \otimes_M m_2)$
2. $\forall m_1, m_2 : \alpha(\gamma(m_1) \otimes_L \gamma(m_2)) = m_1 \otimes_M m_2$ and $\otimes_L : \gamma(M) \times \gamma(M) \to \gamma(M)$

Proof. We will show that the above statements imply each other.

1. First we show that the first one implies the second one

$$\begin{aligned}
\gamma(m_1) \otimes_L \gamma(m_2) &= \gamma(m_1 \otimes_M m_2) \\
\alpha(\gamma(m_1) \otimes_L \gamma(m_2)) &= \alpha(\gamma(m_1 \otimes_M m_2)) \\
\alpha(\gamma(m_1) \otimes_L \gamma(m_2)) &= m_1 \otimes_M m_2
\end{aligned}$$

2. And now in the opposite direction

$$\begin{aligned}
\alpha(\overbrace{\gamma(m_1) \otimes_L \gamma(m_2)}^{=\gamma(m')}) &= m_1 \otimes_M m_2 \\
\alpha(\gamma(m')) &= m_1 \otimes_M m_2 \\
m' &= m_1 \otimes_M m_2 \\
\gamma(m') &= \gamma(m_1 \otimes_M m_2) \\
\gamma(m_1) \otimes_L \gamma(m_2) &= \gamma(m_1 \otimes_M m_2)
\end{aligned}$$

□

A.3 Proof for Lemma 3.14

The set of solutions to the constraint system from Definition 3.13 is a Moore family (i.e., it is closed under $\bigsqcap$), which implies the existence of the least solution.

Proof. Consider a subset $Y^{\mathsf{Q}_\circ}$ of all the solutions of a constraint system $Analysis_F$ for set $\mathsf{Q}_\circ$ of extremal nodes. Every solution $An_F^{\mathsf{Q}_\circ} \in Y$ must satisfy that:

$$An_F^{\mathsf{Q}_\circ}(q) \sqsupseteq \bigoplus\{An_F^{\mathsf{Q}_\circ}(q') \otimes_F \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \in \rightarrow\}$$

Therefore, for every $q' \rightarrow q \in \rightarrow$ a solution in $An_F^{\mathsf{Q}_\circ} \in Y^{\mathsf{Q}_\circ}$ must satisfy:

$$An_F^{\mathsf{Q}_\circ}(q) \sqsupseteq An_F^{\mathsf{Q}_\circ}(q') \otimes_F \mathcal{A}[\![a]\!]$$

Going on, we have:

$$An_F^{\mathsf{Q}_\circ}(q) \sqsupseteq (\bigsqcap Y)(q') \otimes_F \mathcal{A}[\![a]\!]$$

So the right hand side is a lower bound of the set $\{An_F^{\mathsf{Q}_\circ}(q) | An_F^{\mathsf{Q}_\circ} \in Y^{\mathsf{Q}_\circ}\}$, therefore:

$$(\bigsqcap Y)(q) \sqsupseteq (\bigsqcap Y)(q') \otimes_F \mathcal{A}[\![a]\!]$$

But this means that we have:

$$(\bigsqcap Y)(q) \sqsupseteq \bigoplus\{(\bigsqcap Y)(q') \otimes_F \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \in \rightarrow\}$$

Thus the greatest lower bound of a set of solutions is also a solution to the constraint system. Thus the set of all solutions is closed under greatest lower bound. □

A.4 Proof of Proposition 3.16

Consider the *MOP* and *MFP* solutions for an abstract program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, F)$ over a complete flow algebra $(F, \oplus, \otimes, \bar{0}, \bar{1})$, then:

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) \sqsubseteq \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

If the flow algebra is affine and either: $\forall q \in \mathsf{Q} : \mathsf{Path}(\mathsf{Q}_\circ, \{q\}) \neq \emptyset$ or the flow algebra is strict then:

$$MOP_F(\mathsf{Q}_\circ, \mathsf{Q}_\bullet) = \bigoplus_{q \in \mathsf{Q}_\bullet} MFP_F^{\mathsf{Q}_\circ}(q)$$

Proof. First let us consider the first case. It is easy to notice that all we need to prove is that for all $q \in \mathsf{Q}$ the following holds:

$$MOP_F(\mathsf{Q}_\circ, \{q\}) \sqsubseteq MFP_F^{\mathsf{Q}_\circ}(q)$$

Let us define:

$$MOP_F^n(\mathsf{Q}_\circ, \{q\}) = \bigoplus \{\mathcal{A}[\![\pi]\!] \mid \pi \in \mathsf{Path}(\mathsf{Q}_\circ, \{q\}), |\pi| < n\}$$

Clearly we can use MOP_F^n to express MOP:

$$MOP_F(\mathsf{Q}_\circ, \{q\}) = \bigoplus_{n \in \mathbb{N}} MOP_F^n(\mathsf{Q}_\circ, \{q\})$$

Now we can proceed by the induction on n and prove that:

$$MOP_F^n(\mathsf{Q}_\circ, \{q\}) \sqsubseteq MFP_F^{\mathsf{Q}_\circ}(q)$$

The base case is as follows:

$$MOP_F^0(\mathsf{Q}_\circ, \{q\}) = \begin{cases} \bar{1} & \text{if } q \in \mathsf{Q}_\circ \\ \bar{0} & \text{otherwise} \end{cases}$$

Which in both cases is smaller or equal to the $MFP_F^{\mathsf{Q}_\circ}(q)$. Now we can go on to the inductive step and assume that:

$$MOP_F^n(\mathsf{Q}_\circ, \{q\}) \sqsubseteq MFP_F^{\mathsf{Q}_\circ}(q)$$

holds and prove it in case of $n+1$. We consider only one part of the definition where $q \in Q$, the other one is analogous.

$$
\begin{aligned}
&MFP_F^{\mathsf{Q}_\circ}(q) \\
&\quad = \bigoplus \left\{ MFP_F^{\mathsf{Q}_\circ}(q') \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \in \rightarrow \right\} \oplus \bar{1}_F \\
&\quad \sqsupseteq \bigoplus \left\{ MOP_F^{n}(\mathsf{Q}_\circ, \{q'\}) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \in \rightarrow \right\} \oplus \bar{1}_F \\
&\quad = \bigoplus \left\{ (\bigoplus \{\mathcal{A}[\![\pi]\!] \mid \pi \in \mathsf{Path}(\mathsf{Q}_\circ, \{q'\}), |\pi| < n\}) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \rightarrow \right\} \oplus \bar{1}_F \\
&\quad \sqsupseteq \bigoplus \left\{ (\bigoplus \{\mathcal{A}[\![\pi]\!] \otimes \mathcal{A}[\![a]\!] \mid \pi \in \mathsf{Path}(\mathsf{Q}_\circ, \{q'\}), |\pi| < n\}) \mid q' \xrightarrow{a} q \in \rightarrow \right\} \oplus \bar{1}_F \\
&\quad = \bigoplus \left\{ \mathcal{A}[\![\pi]\!] \otimes \mathcal{A}[\![a]\!] \mid \pi \in \mathsf{Path}(\mathsf{Q}_\circ, \{q'\}), |\pi| < n, q' \xrightarrow{a} q \in \rightarrow \right\} \oplus \bar{1}_F \\
&\quad = \bigoplus \{\mathcal{A}[\![\pi']\!] \mid \pi' \in \mathsf{Path}(\mathsf{Q}_\circ, \{q\}), |\pi'| < n+1\} \oplus \bar{1}_F \\
&\quad \sqsupseteq MOP_F^{n+1}(\mathsf{Q}_\circ, \{q\})
\end{aligned}
$$

Now we can go on for the second part of the proof, namely the *MOP* - *MFP* coincidence in the case of affine flow algebra. We also assume that $Path(\mathsf{Q}_\circ, \{q\}) \neq \emptyset$. Note that complete distributivity of the flow algebra is a stronger requirement and can thus be used in place of the above. Then we proceed as follows

$$
\begin{aligned}
MOP_F(\mathsf{Q}_\circ, \{q\}) &= \bigoplus \{\mathcal{A}[\![\pi]\!] \mid \pi \in Path(\mathsf{Q}_\circ, \{q\}\} \\
&= \bigoplus \left\{ \mathcal{A}[\![\pi']\!] \otimes \mathcal{A}[\![a]\!] \mid \pi' \in Path(\mathsf{Q}_\circ, \{q'\}), q' \xrightarrow{a} q \in \rightarrow \right\} \\
&\quad \oplus \bigoplus \{\bar{1} \mid \epsilon \in Path(\mathsf{Q}_\circ, \{q\})\} \\
&= \bigoplus \left\{ (\bigoplus \{\mathcal{A}[\![\pi']\!] \mid \pi' \in Path(\mathsf{Q}_\circ, \{q'\})\}) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \in \rightarrow \right\} \\
&\quad \oplus \bigoplus \{\bar{1} \mid \epsilon \in Path(\mathsf{Q}_\circ, \{q\})\} \\
&= \bigoplus \left\{ MOP_F(\mathsf{Q}_\circ, \{q'\}) \otimes \mathcal{A}[\![a]\!] \mid q' \xrightarrow{a} q \rightarrow \in \rightarrow \right\} \\
&\quad \oplus \bigoplus \{\bar{1} \mid \epsilon \in Path(\mathsf{Q}_\circ, \{q\})\}
\end{aligned}
$$

Which is clearly a solution to the constraints that we defined in the section on the *MFP* solution. But since the *MFP* solution is the least solution for those constraints, therefore we have that:

$$MOP_F(\mathsf{Q}_\circ, \{q\}) \sqsupseteq MFP_F^{\mathsf{Q}_\circ}(q)$$

But from the previous paragraph we know that:

$$MOP_F(\mathsf{Q}_\circ, \{q\}) \sqsubseteq MFP_F^{\mathsf{Q}_\circ}(q)$$

Therefore:

$$MOP_F(\mathsf{Q}_\circ, \{q\}) = MFP_F^{\mathsf{Q}_\circ}(q)$$

□

A.5 Proof of Lemma 3.18

If a program graph $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{B}, M)$ is an upper-approximation of $(\mathsf{Q}, \Sigma, \rightarrow, \mathsf{Q}_I, \mathsf{Q}_F, \mathcal{A}, L)$ by (L, α, γ, M), then for every path π we have that:

$$\mathcal{A}[\![\pi]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![\pi]\!])$$

Proof. First recall that $\mathcal{A}$ and $\mathcal{B}$ are extended to paths in the following way:

$$\mathcal{A}[\![a_1, \ldots, a_n]\!] = \mathcal{A}[\![a_1]\!] \otimes_L \ldots \otimes_L \mathcal{A}[\![a_n]\!]$$
$$\mathcal{B}[\![a_1, \ldots, a_n]\!] = \mathcal{B}[\![a_1]\!] \otimes_M \ldots \otimes_M \mathcal{B}[\![a_n]\!]$$

Now let us consider a path $\pi = a_1, \ldots, a_n$. From Definition 3.17 we know that for each a_i $(1 \leq i \leq n)$ we have:

$$\mathcal{A}[\![a_i]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![a_i]\!])$$

Therefore, from the monotonicity of $\otimes_L$ we have that:

$$\mathcal{A}[\![\pi]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![a_1]\!]) \otimes_L \ldots \otimes_L \gamma(\mathcal{B}[\![a_n]\!])$$

From Definition 3.5 we get the desired result:

$$\mathcal{A}[\![\pi]\!] \sqsubseteq_L \gamma(\mathcal{B}[\![\pi]\!])$$

□

Appendix B

Proofs for Chapter 5

B.1 Soundness proofs

B.1.1 Proof of Theorem 5.4

Consider an automaton $\mathcal{A}$ and its corresponding $\mathcal{A}^{\mathcal{C}}_{pre^*}$ generated by the saturation procedure. Let us assume that we have the least solution λ to the set of constraints $\mathcal{C}$. Then for each pair (p, s) such that $\langle p, s\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_f, \epsilon\rangle$ (where $\sigma \in \Delta^*_{pre}$ and $q_f \in F$), we have $v(\sigma) \sqsubseteq \lambda^*(\rho)$ where $\rho = p \overset{s}{\rightarrow}{}^* q_f$ is in $\mathcal{A}_{pre^*}$.

Proof. Note that we do not need to prove the existence of the paths in the $\mathcal{A}_{pre^*}$ — it is a previously known result [72, 66]. We can use it because our algorithm differs only in the constraint generation, and not in the way new transitions are added. Moreover, as explained above, the additional rules in Δ_{pre} do not change that result.

The proof will proceed by induction on $|\sigma|$ (note that since P and F are disjoint, it is not possible to have $|\sigma| = 0$).

$|\sigma| = 1$ We know that the path in the pushdown system is $\langle p, \gamma\rangle \overset{r}{\Longrightarrow} \langle q_f, \epsilon\rangle$. But this means that $r \in \Delta_{pre} \backslash \Delta$. Existence of $p \overset{\gamma}{\rightarrow} q_f$ follows directly from the

definition of Δ_{pre}. We also have that $f(r) = \bar{1}$. Finally, according to the saturation procedure there exists a constraint: $\bar{1} \sqsubseteq l(p \xrightarrow{\gamma} q_f)$. Therefore, clearly $v([r]) \sqsubseteq \lambda(p \xrightarrow{\gamma} q)$.

$|\sigma| > 1$ In this case we know that the path in the pushdown system is:

$$\langle p, \gamma s_0 \rangle \Longrightarrow \langle q', w s_0 \rangle \overset{\sigma'}{\Longrightarrow}{}^* \langle q_f, \epsilon \rangle$$

for some q', γ, and w. Moreover, $r = \langle p, \gamma \rangle \hookrightarrow \langle q', w \rangle$ where $s = \gamma s_0$.

If $q' \notin P$ then $r \in \Delta_{pre} \setminus \Delta$ and $f(r) = \bar{1}$ (r is one of the added rules to Δ_{pre}). Furthermore, all the rules of σ' must also be in $\Delta_{pre} \setminus \Delta$ and thus there must be a path $\rho = p \xrightarrow{s}{}^* q_f$ in $\mathcal{A}_{pre^*}$ (since it must also be in $\mathcal{A}$). Therefore, $v(\sigma) = \bar{1}$ and for each transition t on the path ρ we have a constraint of the form $\bar{1} \sqsubseteq l(t)$, thus by monotonicity we have $v(\sigma) \sqsubseteq \lambda^*(\rho)$.

Otherwise $q' \in P$ and $r \in \Delta$, so we can use the induction hypothesis to get that:

$$v(\sigma') \sqsubseteq \lambda^*(q' \underset{\rho'}{\xrightarrow{w s_0}}{}^* q_f)$$

where:

$$\rho' = \overbrace{q' \xrightarrow{w}{}^* q''}^{\rho'_1} \underbrace{q'' \xrightarrow{s_0}{}^* q_f}_{\rho'_2}$$

Now the saturation procedure must have added the transition $p \xrightarrow{\gamma} q''$. So we have a path $\rho = p \xrightarrow{\gamma} q'' \xrightarrow{s_0}{}^* q_f$ along with a constraint:

1. if $w = \epsilon$ (so $q' = q''$) the added constraint is

$$f(r) \sqsubseteq l(p \xrightarrow{\gamma} q')$$

2. if $w = \gamma'$ the added constraint is

$$f(r) \otimes l(q' \xrightarrow{\gamma'} q'') \sqsubseteq l(p \xrightarrow{\gamma} q'')$$

3. if $w = \gamma'_1 \gamma'_2$ the added constraint is

$$f(r) \otimes l(q' \xrightarrow{\gamma'_1} q_x) \otimes l(q_x \xrightarrow{\gamma'_2} q'') \sqsubseteq l(p \xrightarrow{\gamma} q'')$$

For case 1 we have:

$$\begin{aligned} v(\sigma) &= f(r) \otimes v(\sigma') \\ &\sqsubseteq f(r) \otimes \lambda^*(q'' \xrightarrow[\rho'_2]{s_0}{}^* q_f) \\ &\sqsubseteq \lambda(p \xrightarrow{\gamma} q') \otimes \lambda^*(q'' \xrightarrow[\rho'_2]{s_0}{}^* q_f) \\ &= \lambda^*(p \xrightarrow[\rho]{s}{}^* q_f) \end{aligned}$$

And for both 2 and 3:

$$\begin{aligned} v(\sigma) &= f(r) \otimes v(\sigma') \\ &\sqsubseteq f(r) \otimes \lambda^*(q' \xrightarrow[\rho'_1]{w}{}^* q'') \otimes \lambda^*(q'' \xrightarrow[\rho'_2]{s_0}{}^* q_f) \\ &\sqsubseteq \lambda^*(p \xrightarrow[q']{\gamma}{}^* \otimes)\lambda^*(q' \xrightarrow[\rho'_1]{w}{}^* q'') \otimes \lambda^*(q'' \xrightarrow[\rho'_2]{s_0}{}^* q_f) \\ &= \lambda^*(p \xrightarrow[\rho]{s}{}^* q_f) \end{aligned}$$

Thus in all possible cases we have that:

$$v(\sigma) \sqsubseteq \lambda^*(p \xrightarrow[\rho]{s}{}^* q_f)$$

□

B.1.2 Proof of Theorem 5.5

Consider an automaton $\mathcal{A}$ and its corresponding $\mathcal{A}^{\mathcal{C}}_{post^*}$ generated by the saturation procedure. Let us assume that we have the least solution λ to the set of constraints $\mathcal{C}$. Then for each pair (p, s) such that $\langle q_f, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p, s\rangle$ (where $\sigma \in \Delta^*_{post}$ and $q_f \in F$), we have $v(\sigma) \sqsubseteq \lambda^*_R(\rho)$ where $\rho = q_f \,{}^*\!\!\xleftarrow{s} p$ is in $\mathcal{A}^{\mathcal{C}}_{post^*}$.

Proof. Note that, as in the case of Pre^*, we do not need to prove the existence of the paths in the $\mathcal{A}_{pre^*}$ — it is a previously known result [72, 66]. Again this is due to the fact that our algorithm differs only in the constraint generation, and not in the way new transitions are added. Moreover, as explained above, the additional rules in Δ_{post} do not change that result.

The proof will proceed by induction on $|\sigma|$ (note that since P and F are disjoint, it is not possible to have $|\sigma| = 0$).

$|\sigma| = 1$ So $s = \gamma$ and we have $\langle q_f, \epsilon\rangle \overset{r}{\Longrightarrow} \langle p, s\rangle$. We know that $r \in \Delta_{post} \setminus \Delta$, and so from the definition of Δ_{post} we have that there is transition $q_f \overset{\gamma}{\leftarrow} p$ and $v([r]) = \bar{1}$. Moreover, from the saturation procedure we have a constraint $\bar{1} \sqsubseteq h(q_f \overset{\gamma}{\leftarrow} p)$. Therefore, $v([r]) \sqsubseteq \lambda(q_f \overset{\gamma}{\leftarrow} p)$.

$|\sigma| > 1$ So we have:

$$\langle q_f, \epsilon\rangle \overset{\sigma'}{\Longrightarrow}{}^{*} \langle q', s'\rangle \overset{r}{\Longrightarrow} \langle p, s\rangle$$

where $\sigma = \sigma' r$.

If $q' \notin P$ then $r \in \Delta_{post} \setminus \Delta$ and it must be of the form $r = \langle q', \epsilon\rangle \hookrightarrow \langle p, \gamma\rangle$ where $s = \gamma s'$ (r is one of the additional rules to the Δ_{post}). But that means that all the remaining rules in σ' must also be one of those additional rules ($\Delta_{post} \setminus \Delta$). Thus the weight of every transition t on the path $q_f \, {}^{*}\!\overset{s}{\leftarrow} p$ is $\lambda(t) = \bar{1}$ (its existence follows directly from the definition of Δ_{post}). Moreover, all of them must have a corresponding constraint of the form $\bar{1} \sqsubseteq h(t)$. Therefore, by monotonicity we have $\bar{1} \sqsubseteq \lambda_R^*(q_f \, {}^{*} \underset{\rho}{\overset{s}{\leftarrow}} p)$ and so $v(\sigma) \sqsubseteq \lambda_R^*(q_f \, {}^{*} \underset{\rho}{\overset{s}{\leftarrow}} p)$.

Otherwise $q' \in P$ and $r \in \Delta$, $r = \langle q', \gamma'\rangle \hookrightarrow \langle p, w\rangle$ and $s = ws_0$, $s' = \gamma' s_0$. Since $|\sigma'| < |\sigma|$ we can use the induction hypothesis to get that:

$$v(\sigma') \sqsubseteq \lambda_R^*(q_f \, {}^{*} \underset{\rho'}{\overset{s'}{\leftarrow}} q')$$

where:

$$\rho' = q_f \, {}^{*}\!\overset{s'}{\leftarrow} q' = \overbrace{q_f \quad {}^{*}\!\overset{s_0}{\leftarrow} \quad q''}^{\rho'_2} \underbrace{\overset{\gamma'}{\dashleftarrow} \quad q'}_{\rho'_1}$$

for some q''. And so we have three possibilities, depending on w:

1. if $w = \epsilon$, the transition $q'' \overset{\epsilon}{\leftarrow} p$ along with the following constraint:

$$h^{\epsilon}(q'' \underset{\rho'_1}{\overset{\gamma'}{\dashleftarrow}} q') \otimes f(r) \sqsubseteq h(q'' \overset{\epsilon}{\leftarrow} p)$$

Therefore, the solution will have to satisfy:

$$\lambda_R^*(q'' \, {}^{*} \underset{\rho'_1}{\overset{\gamma'}{\leftarrow}} q') \otimes f(r) \sqsubseteq \lambda(q'' \overset{\epsilon}{\leftarrow} p)$$

and so:

$$\begin{aligned}
v(\sigma) &= v(\sigma') \otimes f(r) \\
&\sqsubseteq \lambda_R^*(q_f \ {}^* \xleftarrow[\rho']{s'} q') \otimes f(r) \\
&= \lambda_R^*(q_f \ {}^* \xleftarrow[\rho'_2]{s_0} q'') \otimes \lambda_R^*(q'' \ {}^* \xleftarrow[\rho'_1]{\gamma'} q') \otimes f(r) \\
&\sqsubseteq \lambda_R^*(q_f \ {}^* \xleftarrow[\rho'_2]{s_0} q'') \otimes \lambda(q'' \xleftarrow{\epsilon} p) \\
&= \lambda_R^*(q_f \ {}^* \xleftarrow[\rho]{s} p)
\end{aligned}$$

2. if $w = \gamma$, the transition $q'' \xleftarrow{\gamma} p$ along with the following constraint:

$$h^\epsilon(q'' \xleftarrow[\rho'_1]{\gamma'} q') \otimes f(r) \sqsubseteq h(q'' \xleftarrow{\gamma} p)$$

Therefore, the solution will have to satisfy:

$$\lambda_R^*(q'' \ {}^* \xleftarrow[\rho'_1]{\gamma'} q') \otimes f(r) \sqsubseteq \lambda(q'' \xleftarrow{\gamma} p)$$

and so:

$$\begin{aligned}
v(\sigma) &= v(\sigma') \otimes f(r) \\
&\sqsubseteq \lambda_R^*(q_f \ {}^* \xleftarrow[\rho']{s'} q') \otimes f(r) \\
&= \lambda_R^*(q_f \ {}^* \xleftarrow[\rho'_2]{s_0} q'') \otimes \lambda_R^*(q'' \ {}^* \xleftarrow[\rho'_1]{\gamma'} q') \otimes f(r) \\
&\sqsubseteq \lambda_R^*(q_f \ {}^* \xleftarrow[\rho'_2]{s_0} q'') \otimes \lambda(q'' \xleftarrow{\gamma} p) \\
&= \lambda_R^*(q_f \ {}^* \xleftarrow[\rho]{s} p)
\end{aligned}$$

3. if $w = \gamma_1\gamma_2$, the transitions $q_{p,\gamma_1} \xleftarrow{\gamma_1} q'$ and $q'' \xleftarrow{\gamma_2} q_{p,\gamma_1}$ along with the following constraints:

$$\bar{1} \sqsubseteq h(q_{p,\gamma_1} \xleftarrow{\gamma_1} q)$$

and:

$$h^\epsilon(q'' \xleftarrow[\rho'_1]{\gamma'} q') \otimes f(r) \sqsubseteq h(q'' \xleftarrow{\gamma_2} q_{p,\gamma_1})$$

Therefore, the solution will have to satisfy:

$$\bar{1} \sqsubseteq \lambda(q_{p,\gamma_1} \xleftarrow{\gamma_1} q')$$

$$\lambda_R^*(q'' * \underset{\rho_1'}{\overset{\gamma'}{\leftarrow}} q') \otimes f(r) \sqsubseteq \lambda(q'' \overset{\gamma_2}{\leftarrow} q_{p,\gamma_1})$$

and so:

$$\begin{aligned}
v(\sigma) &= v(\sigma') \otimes f(r) \\
&\sqsubseteq \lambda_R^*(q_f * \underset{\rho'}{\overset{s'}{\leftarrow}} q') \otimes f(r) \\
&= \lambda_R^*(q_f * \underset{\rho_2'}{\overset{s_0}{\leftarrow}} q'') \otimes \lambda_R^*(q'' * \underset{\rho_1'}{\overset{\gamma'}{\leftarrow}} q') \otimes f(r) \\
&\sqsubseteq \lambda_R^*(q_f * \underset{\rho_2'}{\overset{s_0}{\leftarrow}} q'') \otimes \lambda(q'' \overset{\gamma}{\leftarrow} p) \\
&= \lambda_R^*(q_f * \underset{\rho_2'}{\overset{s_0}{\leftarrow}} q'') \otimes \lambda(q'' \overset{\gamma_2}{\leftarrow} q_{p,\gamma_1}) \otimes \lambda(q_{p,\gamma_1} \overset{\gamma_1}{\leftarrow} q') \\
&= \lambda_R^*(q_f * \underset{\rho}{\overset{s}{\leftarrow}} p)
\end{aligned}$$

□

B.2 Continuity proof (Lemma 5.6)

The function F, defined as:

$$\begin{aligned}
&F : (\delta \to D) \to (\delta \to D) \\
&F(m)t = \bigoplus_{c \in \mathcal{C}_t} lhs_m(c)
\end{aligned}$$

is continuous, i.e, for any non-empty chain Y:

$$F(\bigsqcup Y) = \bigsqcup_{m \in Y} F(m)$$

Proof. Since we are assuming that D is a complete lattice and m is a total function, then $\delta \to D$ defines a complete lattice as well. Furthermore, we have that for any $Y \subseteq \delta \to D$:

$$(\bigsqcup Y)t = \bigoplus_{m \in Y} m(t) \tag{B.1}$$

Therefore, we have:

$$
\begin{aligned}
& F(\bigsqcup Y)t \\
& \qquad = [\text{ definition of } F \text{ }] \\
& \bigoplus\{lhs_{\bigsqcup Y}(c) \mid c \in \mathcal{C}_t\} \\
& \qquad = [\text{ equation (B.1) }] \\
& \bigoplus\{lhs_{\lambda t'.\bigoplus_{m\in Y} m(t')}(c) \mid c \in \mathcal{C}_t\} \\
& \qquad = [\text{ } D \text{ is affine, } Y \text{ is not empty and the constraints are finite }] \\
& \bigoplus\{\bigoplus_{m\in Y} lhs_m(c) \mid c \in \mathcal{C}_t\} \\
& \qquad = [\text{ } D \text{ is a complete lattice }] \\
& \bigoplus_{m\in Y}(\bigoplus\{lhs_m(c) \mid c \in \mathcal{C}_t\}) \\
& \qquad = [\text{ definition of } F \text{ }] \\
& \bigoplus_{m\in Y} F(m)t \\
& \qquad = [\text{ equation (B.1) }] \\
& (\bigsqcup_{m\in Y} F(m))t
\end{aligned}
$$

□

B.3 Completeness proofs

B.3.1 Proof of Lemma 5.7

For every transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}_{pre^*}$ there exists a sequence $\sigma \in \Delta_{pre}$ such that $\langle q, \gamma\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon\rangle$.

Proof. Proof will proceed by induction on $\mathcal{A}_i$, where $\mathcal{A}_i$ corresponds to the initial automaton after i steps of the saturation procedure.

$i = 0$ Follows from the definition of Δ_{pre}.

$i > 0$ We assume the property holds for $\mathcal{A}_i$ and prove it for $\mathcal{A}_{i+1}$. Consider that the saturation procedure adds a transition $p_s \xrightarrow{\gamma} q_d$ (note that the saturation procedure works on Δ) because of:

- A pushdown rule $r = \langle p_s, \gamma \rangle \hookrightarrow \langle q_d, \epsilon \rangle$. The result is immediate from the rule.
- A pushdown rule $r = \langle p_s, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle$ and a transition $p' \xrightarrow{\gamma'} q_d$ in $\mathcal{A}_i$. We use the induction hypothesis on $p' \xrightarrow{\gamma'} q_d$ and get that there exists σ such that $\langle p', \gamma' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_d, \epsilon \rangle$. But then we also have that:

$$\langle p_s, \gamma \rangle \overset{r}{\Longrightarrow} \langle p', \gamma' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_d, \epsilon \rangle$$

- A pushdown rule $r = \langle p_s, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle$ and a path $p' \xrightarrow{\gamma'} q'' \xrightarrow{\gamma''} q_d$ in $\mathcal{A}_i$. We use the induction hypothesis on $p' \xrightarrow{\gamma'} q''$ and $q'' \xrightarrow{\gamma''} q_d$ to get that there exists σ' and σ'' such that $\langle p', \gamma' \rangle \overset{\sigma'}{\Longrightarrow}{}^* \langle q'', \epsilon \rangle$ and $\langle q'', \gamma' \rangle \overset{\sigma''}{\Longrightarrow}{}^* \langle q_d, \epsilon \rangle$. And again we have that:

$$\langle p_s, \gamma \rangle \overset{r}{\Longrightarrow} \langle q', \gamma'\gamma'' \rangle \overset{\sigma'\sigma''}{\Longrightarrow}{}^* \langle q_d, \epsilon \rangle$$

□

B.3.2 Proof of Lemma 5.8

Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$ where $\mathcal{F}$ is affine and an automaton $\mathcal{A}^{\mathcal{C}}_{pre^*}$ created by the saturation procedure. For every transition $q \xrightarrow{\gamma} q'$ in this automaton we have that:

$$\lambda(q \xrightarrow{\gamma} q') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle, \sigma \in \Delta^*_{pre}\}$$

Proof. Let us also denote by $\mathcal{A}^{\mathcal{C}}_i$ the automaton $\mathcal{A}$ after i steps of the saturation procedure. Also let us denote the least solution for $\mathcal{A}^{\mathcal{C}}_i$ by λ_i. We will prove by induction on i that for every transition $q \xrightarrow{\gamma} q'$ in $\mathcal{A}^{\mathcal{C}}_i$ we have that:

$$\lambda_i(q \xrightarrow{\gamma} q') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle, \sigma \in \Delta^*_{pre}\}$$

$i = 0$ $\mathcal{A}^{\mathcal{C}}_0$ is just the initial automaton $\mathcal{A}$ with the set $\mathcal{C}$ containing one constraint for every transition of $\mathcal{A}$. The property clearly holds.

$i > 0$ We assume the property holds for $\mathcal{A}_i^{\mathcal{C}}$ and prove it for $\mathcal{A}_{i+1}^{\mathcal{C}}$, i.e., prove that adding a constraint (and maybe a transition as well) preserves the property of interest.

Let t be the transition that the added constraint refers to. Observe that if t was already in the automaton $\mathcal{A}_i^{\mathcal{C}}$, then it is possible that $\lambda(t)$ might be on the left-hand side of some other constraint. Therefore, the least solution for the new set of constraints might be different for other transitions as well; in other words the value/information from the new constraint might have to be propagated throughout other constraints to get λ_{i+1}. Now let λ_i^j denote the solution after j steps of fixed point computation with the new constraint, starting with:

$$\lambda_i^0(t) = \begin{cases} \bar{0} & \text{if } t \text{ was added} \\ \lambda_i(t) & \text{otherwise } (t \text{ was in } \mathcal{A}_i^{\mathcal{C}}) \end{cases}$$

Using induction on j we will prove that the property of interest is maintained by the computation.

Note that we can use here Kleene iteration due to Lemma 5.6.

$j = 0$ Immediate from outer induction hypothesis.

$j > 0$ In the following we will use the fact that the flow algebra is affine; it is enough for our purposes because from Lemma 5.7 it follows that the sets (of pushdown paths) on the right-hand sides are not empty. Let us consider each form of the possible constraints:

- $f(r) \sqsubseteq \lambda(q \xrightarrow{\gamma} q')$ where $r = \langle q, \gamma \rangle \hookrightarrow \langle q', \epsilon \rangle$. We know that:

$$\lambda_i^{j+1}(q \xrightarrow{\gamma} q') = \lambda_i^j(q \xrightarrow{\gamma} q') \oplus f(r)$$

  Moreover, from the rule r it immediately follows that:

$$f(r) \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\}$$

  Using this and the induction hypothesis on $\lambda_i^j(q \xrightarrow{\gamma} q')$:

$$\lambda_i^{j+1}(q \xrightarrow{\gamma} q') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\}$$

- $f(r) \otimes \lambda(q'' \xrightarrow{\gamma''} q') \sqsubseteq \lambda(q \xrightarrow{\gamma} q')$ where $r = \langle q, \gamma \rangle \hookrightarrow \langle q'', \gamma'' \rangle$ and $q'' \xrightarrow{\gamma''} q'$. We have that:

$$\lambda_i^{j+1}(q \xrightarrow{\gamma} q') = \lambda_i^j(q \xrightarrow{\gamma} q') \oplus (f(r) \otimes \lambda_i^j(q'' \xrightarrow{\gamma''} q'))$$

  Now let us use the induction hypothesis:

$$\lambda_i^j(q'' \xrightarrow{\gamma''} q') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q'', \gamma'' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\}$$

Multiplying both sides by $f(r)$ and using that $\otimes$ is affine:

$$\begin{aligned} f(r) \otimes \lambda_i^j(q'' \xrightarrow{\gamma''} q') &\sqsubseteq \bigoplus\{f(r) \otimes v(\sigma) \mid \langle q'', \gamma'' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\} \\ &\sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\} \end{aligned}$$

Thus:

$$\lambda_i^{j+1}(q \xrightarrow{\gamma} q') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\}$$

- $f(r) \otimes \lambda(q'' \xrightarrow{\gamma_1''} q_1') \otimes \lambda(q_1' \xrightarrow{\gamma_2''} q') \sqsubseteq \lambda(q \xrightarrow{\gamma} q')$ where $r = \langle q, \gamma \rangle \hookrightarrow \langle q'', \gamma_1'' \gamma_2'' \rangle$ and $q'' \xrightarrow{\gamma_1''} q_1' \xrightarrow{\gamma_2''} q'$. We have that:

$$\begin{aligned} \lambda_i^{j+1}(q \xrightarrow{\gamma} q') = \lambda_i^j(q \xrightarrow{\gamma} q') \\ \oplus (f(r) \otimes \lambda_i^j(q'' \xrightarrow{\gamma_1''} q_1') \otimes \lambda_i^j(q_1' \xrightarrow{\gamma_2''} q')) \end{aligned}$$

We use the induction hypothesis twice to get:

$$\begin{aligned} \lambda_i^j(q'' \xrightarrow{\gamma_1''} q_1') &\sqsubseteq \bigoplus\{v(\sigma) \mid \langle q'', \gamma_1'' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_1', \epsilon \rangle\} \\ \lambda_i^j(q_1' \xrightarrow{\gamma_2''} q') &\sqsubseteq \bigoplus\{v(\sigma) \mid \langle q_1', \gamma_2'' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\} \end{aligned}$$

From monotonicity and the fact that $\otimes$ is affine we get that:

$$\begin{aligned} & f(r) \otimes \lambda_i^j(q'' \xrightarrow{\gamma_1''} q_1') \otimes \lambda_i^j(q_1' \xrightarrow{\gamma_2''} q') \\ & \quad \sqsubseteq \bigoplus\{f(r) \otimes v(\sigma_1) \otimes v(\sigma_2) \mid \langle q'', \gamma_1'' \rangle \overset{\sigma_1}{\Longrightarrow}{}^* \langle q_1', \epsilon \rangle, \langle q_1', \gamma_2'' \rangle \overset{\sigma_2}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\} \\ & \quad \sqsubseteq \bigoplus\{f(r) \otimes v(\sigma) \mid \langle q'', \gamma_1'' \gamma_2'' \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\} \\ & \quad \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\} \end{aligned}$$

Thus:

$$\lambda_i^{j+1}(q \xrightarrow{\gamma} q') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \gamma \rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q', \epsilon \rangle\}$$

$\square$

B.3.3 Proof of Lemma 5.9

Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$ where $\mathcal{F}$ is affine and a $\mathcal{A}^{\mathcal{C}}_{pre^*}$ automaton created by the saturation procedure. For every path $\rho = q \xrightarrow{s} {}^* q'$ in this automaton we have that:

$$\lambda^*(q \xrightarrow[\rho]{s} {}^* q') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, s\rangle \xRightarrow{\sigma}{}^* \langle q', \epsilon\rangle, \sigma \in \Delta^*_{pre}\}$$

Proof. The proof will proceed with the induction on the number of transitions $|\rho|$ (we will use the inductive definition of λ^*).

$|\rho| = 1$ So ρ is just a single transition, therefore according to the definition of λ we have:

$$\lambda^*(q \xrightarrow[\rho]{s} {}^* q') = \lambda(q \xrightarrow{s} q')$$

The result follows from Lemma 5.8.

$1 < |\rho|$ Again using the definition of λ we have:

$$\lambda^*(q \xrightarrow[\rho]{s} {}^* q') = \lambda(q \xrightarrow{\gamma} q'') \otimes \lambda^*(q'' \xrightarrow[\rho']{s'} {}^* q')$$

where $s = \gamma s'$, $q'' \in Q$, and:

$$\rho = q \xrightarrow{\gamma} \underbrace{q'' \xrightarrow{s'}{}^* q'}_{\rho'}$$

Now we can use Lemma 5.8 again and the induction hypothesis (since $|\rho|' < |\rho|$) to get:

$$\lambda(q \xrightarrow{\gamma} q'') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, \gamma\rangle \xRightarrow{\sigma}{}^* \langle q'', \epsilon\rangle, \sigma \in \Delta_{pre}\}$$

$$\lambda^*(q'' \xrightarrow[\rho']{s'} {}^* q') \sqsubseteq \bigoplus \{v(\sigma) \mid \langle q'', s'\rangle \xRightarrow{\sigma}{}^* \langle q', \epsilon\rangle, \sigma \in \Delta_{pre}\}$$

Finally, we use the fact that the flow algebra is affine:

$$\begin{aligned}
&\lambda^*(q \xrightarrow[\rho]{s} {}^* q') \\
&\sqsubseteq \bigoplus \{v(\sigma) \otimes v(\sigma') \mid \langle q, \gamma\rangle \xRightarrow{\sigma}{}^* \langle q'', \epsilon\rangle, \langle q'', s'\rangle \xRightarrow{\sigma'}{}^* \langle q', \epsilon\rangle, \sigma, \sigma' \in \Delta_{pre}\} \\
&\sqsubseteq \bigoplus \{v(\sigma) \mid \langle q, s\rangle \xRightarrow{\sigma}{}^* \langle q', \epsilon\rangle, \sigma \in \Delta_{pre}\}
\end{aligned}$$

□

B.3.4 Proof of Theorem 5.10

Consider an automaton $\mathcal{A}^{\mathcal{C}}_{pre^*}$ constructed by the saturation procedure and the least solution λ to the set of its constraints $\mathcal{C}$. If the flow algebra is affine then for every path $\rho = p \xrightarrow{s}{}^* q_f$ where $q_f \in F$ we have that:

$$\lambda^*(p \underset{\rho}{\xrightarrow{s}}{}^* q_f) = \bigoplus\{v(\sigma) \mid \langle p, s\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q_f, \epsilon\rangle, \sigma \in \Delta^*_{pre}\}$$

Proof. The result follows directly from Theorem 5.4 and Lemma 5.9. □

B.3.5 Proof of Lemma 5.11

For every transition $q' \xleftarrow{\gamma_\epsilon} q$ ($\gamma_\epsilon \in \Gamma \cup \{\epsilon\}$) in $\mathcal{A}_{post^*}$ there exists a sequence σ of pushdown rules in $\Delta_{post\text{-}2}$ such that $\langle q', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, \gamma_\epsilon\rangle$.

Proof. Let us denote by $\mathcal{A}_i$ the automaton $\mathcal{A}$ after i steps of the saturation procedure. Proof will proceed by induction on i.

$i = 0$ Follows from the definition of $\Delta_{post\text{-}2}$.

$i > 0$ We assume the property holds for $\mathcal{A}_i$ and prove it for $\mathcal{A}_{i+1}$. Consider that the saturation procedure[1]

- Adds a transition $q_d \xleftarrow{\epsilon} p_s$ because of a pushdown rule $r = \langle p', \gamma'\rangle \hookrightarrow \langle p_s, \epsilon\rangle$ and a path $q_d \overset{\gamma'}{\dashleftarrow} p'$. We can use the induction hypothesis to get that there exists σ such that $\langle q_d, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p', \gamma'\rangle$. But then clearly $\langle q_d, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p', \gamma'\rangle \overset{r}{\Longrightarrow} \langle p, \epsilon\rangle$.
- Adds a transition $q_d \xleftarrow{\gamma} p_s$ because of a pushdown rule $r = \langle p', \gamma'\rangle \hookrightarrow \langle p_s, \epsilon\rangle$ and a path $q_d \overset{\gamma'}{\dashleftarrow} p'$. We can use the induction hypothesis to get that there exists σ such that $\langle q_d, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p', \gamma'\rangle$. Again it is clear that $\langle q_d, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p', \gamma'\rangle \overset{r}{\Longrightarrow} \langle p, \epsilon\rangle$.
- Adds transitions $q_{p_s,\gamma_1} \xleftarrow{\gamma_1} p_s$ and $q_d \xleftarrow{\gamma_2} q_{p_s,\gamma_1}$ because of a pushdown rule $r = \langle p', \gamma'\rangle \hookrightarrow \langle p_s, \gamma_1\gamma_2\rangle$ and a path $q_d \overset{\gamma'}{\dashleftarrow} p'$. According to the definition of $\Delta_{post\text{-}2}$ we know that there are $r_1 = \langle p', \gamma'\rangle \hookrightarrow$

[1] Note that the saturation procedure works on Δ.

$\langle q_{p_s,\gamma_1}, \gamma_2\rangle$ and $r_2 = \langle q_{p_s,\gamma_1}, \epsilon\rangle \hookrightarrow \langle p_s, \gamma_1\rangle$. So we immediately have the path for the first transition:

$$\langle q_{p_s,\gamma_1}, \epsilon\rangle \xRightarrow{r_2} \langle p_s, \gamma_1\rangle$$

Moreover, we can use the induction hypothesis to get that there exists σ such that $\langle q_d, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p', \gamma'\rangle$ and so we also have that:

$$\langle q_d, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p', \gamma'\rangle \xRightarrow{r_1} \langle q_{p_s,\gamma_1}, \gamma_2\rangle$$

□

B.3.6 Proof of Lemma 5.12

Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$ where $\mathcal{F}$ is affine and an automaton $\mathcal{A}^{\mathcal{C}}_{post^*}$ created by the saturation procedure. For every transition $q' \xleftarrow{\gamma_\epsilon} q$ ($\gamma_\epsilon \in \Gamma \cup \{\epsilon\}$) in this automaton we have that:

$$\lambda(q' \xleftarrow{\gamma_\epsilon} q) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \xRightarrow{\sigma}{}^* \langle q, \gamma_\epsilon\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

Proof. Let us denote by $\mathcal{A}^{\mathcal{C}}_i$ the automaton $\mathcal{A}^{\mathcal{C}}$ after i steps of saturation procedure and similarly the least solution for it by λ_i. We will prove by induction on i that for every transition $q' \xleftarrow{\gamma_\epsilon} q$ we have that:

$$\lambda_i(q' \xleftarrow{\gamma_\epsilon} q) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \xRightarrow{\sigma}{}^* \langle q, \gamma_\epsilon\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

$i = 0$ The only constraints are of the form $\bar{1} \sqsubseteq l(t)$ where t is a transition in $\mathcal{A}$. Therefore, the least solution for each t is $\lambda_i(t) = \bar{1}$. We also know that for every $r \in \Delta_{post\text{-}2} \setminus \Delta$, $f(r) = \bar{1}$. So the right hand side is at least $\bar{1}$. Thus our property holds.

$i > 0$ We assume the property holds for $\mathcal{A}^{\mathcal{C}}_i$ and prove it for $\mathcal{A}^{\mathcal{C}}_{i+1}$, i.e., prove that adding a constraint (and maybe a transition as well) preserves the property of interest.

Let t bi the transition that the added constraint refers to. Observe that if t was already in the automaton $\mathcal{A}^{\mathcal{C}}_i$, then it is possible that $h(t)$ might be on the left-hand side of some other constraint. Therefore, the least solution for the new set of constraints might be different for other transitions as well; in other words the value/information from the new constraint might

have to be propagated throughout other constraints to get λ_{i+1}. Now let λ_i^j denote the solution after j steps of fixed point computation with the new constraint, starting with:

$$\lambda_i^0(t) = \begin{cases} \bar{0} & \text{if } t \text{ was added} \\ \lambda_i(t) & \text{otherwise } (t \text{ was in } \mathcal{A}_i^{\mathcal{C}}) \end{cases}$$

Using induction on j we will prove that the property is maintained by the computation.

Note that we can use here Kleene iteration due to Lemma 5.6.

$j = 0$ Immediate from outer induction hypothesis.

$j > 0$ We assume the property hold for λ_i^j and prove that it also holds for λ_i^{j+1}. In the following we use the fact that the flow algebra is affine, this is enough since from Lemma 5.11 it follows that the sets (of pushdown paths) on the right hand sides are not empty. Let us consider three possibilities of constraints:

- if the constraint is:

$$h(q \xleftarrow{\gamma'} p') \otimes f(r) \sqsubseteq h(q \xleftarrow{\epsilon} p)$$

or:

$$h(q \xleftarrow{\gamma'} q'') \otimes h(q'' \xleftarrow{\epsilon} p') \otimes f(r) \sqsubseteq h(q \xleftarrow{\epsilon} p)$$

where $r = \langle p', \gamma' \rangle \hookrightarrow \langle p, \epsilon \rangle \in \Delta$. Let us only consider the more complex case with additional ϵ transition (the one without is similar). We need to calculate the value of $\lambda_i^{j+1}(q \xleftarrow{\epsilon} p)$ — it should be its old value combined with the new one:

$$\lambda_i^{j+1}(q \xleftarrow{\epsilon} p) = \lambda_i^j(q \xleftarrow{\epsilon} p) \oplus \left(\lambda_i^j(q \xleftarrow{\gamma'} q'') \otimes \lambda_i^j(q'' \xleftarrow{\epsilon} p') \otimes f(r) \right)$$

Let us use the induction hypothesis (inner induction) three times to get:

$$\lambda_i^j(q \xleftarrow{\epsilon} p) \sqsubseteq \bigoplus \{ v(\sigma) \mid \langle q, \epsilon \rangle \xRightarrow{\sigma}{}^* \langle p, \epsilon \rangle, \sigma \in \Delta^*_{post\text{-}2} \}$$

$$\lambda_i^j(q \xleftarrow{\gamma'} q'') \sqsubseteq \bigoplus \{ v(\sigma) \mid \langle q, \epsilon \rangle \xRightarrow{\sigma}{}^* \langle q'', \gamma' \rangle, \sigma \in \Delta^*_{post\text{-}2} \}$$

$$\lambda_i^j(q'' \xleftarrow{\epsilon} p') \sqsubseteq \bigoplus \{ v(\sigma) \mid \langle q'', \epsilon \rangle \xRightarrow{\sigma}{}^* \langle p', \epsilon \rangle, \sigma \in \Delta^*_{post\text{-}2} \}$$

Using the above and the fact that our flow algebra is affine, we get:

$$\begin{aligned}
&\lambda_i^j(q \xleftarrow{\gamma'} q'') \otimes \lambda_i^j(q'' \xleftarrow{\epsilon} p') \otimes f(r) \\
&\quad \sqsubseteq \bigoplus\{v(\sigma_1) \otimes v(\sigma_2) \otimes f(r) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma_1}{}^* \langle q'', \gamma'\rangle, \\
&\qquad\qquad \langle q'', \epsilon\rangle \xRightarrow{\sigma_2}{}^* \langle p', \epsilon\rangle, \\
&\qquad\qquad \langle p', \gamma'\rangle \xRightarrow{r} \langle p, \epsilon\rangle, \\
&\qquad\qquad \sigma \in \Delta^*_{post\text{-}2}\} \\
&\quad \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p, \epsilon\rangle, \sigma \in \Delta^*_{post\text{-}2}\}
\end{aligned}$$

Now since $\bigoplus$ gives the least upper bound, we have that:

$$\lambda_i^{j+1}(q \xleftarrow{\epsilon} p) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p, \epsilon\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

- if the constraint is:

$$h(q \xleftarrow{\gamma'} p') \otimes f(r) \sqsubseteq h(q \xleftarrow{\gamma} p)$$

  or:

$$h(q \xleftarrow{\gamma'} q'') \otimes h(q'' \xleftarrow{\epsilon} p') \otimes f(r) \sqsubseteq h(q \xleftarrow{\gamma} p)$$

  where r must be $r = \langle p', \gamma'\rangle \hookrightarrow \langle p, \gamma\rangle \in \Delta$. The case is analogous to the previous one (we just have γ instead of ϵ).
- if the constraint is one of:

$$\bar{1} \sqsubseteq h(q_{p,\gamma_1} \xleftarrow{\gamma_1} p)$$

  or:

$$h(q \xleftarrow{\gamma'} q'') \otimes h(q'' \xleftarrow{\epsilon} p') \otimes f(r) \sqsubseteq h(q \xleftarrow{\gamma_2} q_{p,\gamma_1})$$

  (alternatively without the ϵ-transition:

$$h(q \xleftarrow{\gamma'} p') \otimes f(r) \sqsubseteq h(q \xleftarrow{\gamma_2} q_{p,\gamma_1})$$

  but we will only consider the former, since it is a bit more complex and the proof for the latter is almost the same).
  We know that $r = \langle p', \gamma'\rangle \hookrightarrow \langle p, \gamma_1\gamma_2\rangle \in \Delta$ and so that we have $r_1, r_2 \in \Delta_{post\text{-}2}$ such that $r_1 = \langle p', \gamma'\rangle \hookrightarrow \langle q_{p,\gamma_1}, \gamma_2\rangle$ and $r_2 = \langle q_{p,\gamma_1}, \epsilon\rangle \hookrightarrow \langle p, \gamma_1\rangle$ with $f(r_1) = f(r)$ and $f(r_2) = \bar{1}$.
  For the first trivial inequality the property is clearly preserved. Let us focus on the second one. We know that:

$$\begin{aligned}
\lambda_i^{j+1}(q \xleftarrow{\gamma_2} q_{p,\gamma_2}) &= \lambda_i^j(q \xleftarrow{\gamma_2} q_{p,\gamma_2}) \\
&\quad \oplus \left(\lambda_i^j(q \xleftarrow{\gamma'} q') \otimes \lambda_i^j(q' \xleftarrow{\epsilon} p') \otimes f(r)\right) \qquad \text{(B.2)}
\end{aligned}$$

for some $q' \in Q$. Using induction hypothesis we have that:

$$\lambda_i^j(q \xleftarrow{\gamma_2} q_{p,\gamma_1}) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle q_{p,\gamma_1}, \gamma_2\rangle\} \tag{B.3}$$

$$\lambda_i^j(q \xleftarrow{\gamma'} q') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle q', \gamma'\rangle\}$$

$$\lambda_i^j(q' \xleftarrow{\epsilon} p') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p', \epsilon\rangle\}$$

Using the last two and the fact that the flow algebra is affine, we get the following:

$$\begin{aligned}
&\lambda_i^j(q \xleftarrow{\gamma'} q') \otimes \lambda_i^j(q' \xleftarrow{\epsilon} p') \otimes f(r) \\
&\sqsubseteq \{v(\sigma_1) \otimes v(\sigma_2) \otimes f(r) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle q', \gamma'\rangle, \langle q', \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p', \epsilon\rangle\} \\
&\sqsubseteq \{v(\sigma) \otimes f(r_1) \otimes f(r_2) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p', \gamma'\rangle \xRightarrow{r_1} \langle q_{p,\gamma_1}, \gamma_2\rangle \xRightarrow{r_2} \langle p, \gamma_1\gamma_2\rangle\} \\
&\sqsubseteq \{v(\sigma) \mid \langle q, \epsilon\rangle \xRightarrow{\sigma}{}^* \langle p, \gamma_1\gamma_2\rangle\}
\end{aligned}$$

So from this and (B.2) and (B.3) we have the desired result.

□

B.3.7 Proof of Lemma 5.13

Consider a weighted pushdown system $\mathcal{W} = (\mathcal{P}, \mathcal{F}, f)$ where $\mathcal{F}$ is affine and a $\mathcal{A}^{\mathcal{C}}_{post^*}$ automaton created by the saturation procedure. For every path $\rho = q' \xleftarrow{s} q$ ($s \in \Gamma^*$) in this automaton we have that:

$$\lambda_R^*(q' \overset{s}{\underset{\rho}{\leftarrow}} q) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \xRightarrow{\sigma}{}^* \langle q, s\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

Proof. The proof will proceed with the induction on the number of transitions in ρ (we will use the inductive definition of λ).

$|\rho| = 1$ According to the definition of λ we have:

$$\lambda_R^*(q' \overset{s}{\underset{\rho}{\leftarrow}} q) = \lambda(q' \xleftarrow{\gamma_\epsilon} q)$$

The result follows from Lemma 5.12.

$|\rho| > 1$ Again using the definition of λ_R^* we have:

$$\lambda_R^*(q' \overset{*}{\underset{\rho}{\xleftarrow{s}}} q) = \lambda_R^*(q' \overset{*}{\underset{\rho'}{\xleftarrow{s'}}} q'') \otimes \lambda(q'' \xleftarrow{\gamma_\epsilon} q)$$

where $s = \gamma_\epsilon s'$, $q'' \in Q$, and:

$$\rho = \underbrace{q' \overset{*}{\xleftarrow{s'}} q''}_{\rho'} \xleftarrow{\gamma_\epsilon} q$$

Now we can use the Lemma 5.12 along with the induction hypothesis (since $|\rho| > |\rho'|$) to get:

$$\lambda(q'' \xleftarrow{\gamma_\epsilon} q) \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q'', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, \gamma_\epsilon\rangle, \sigma \in \Delta_{post\text{-}2}\}$$

$$\lambda_R^*(q' \overset{*}{\underset{\rho'}{\xleftarrow{s'}}} q'') \sqsubseteq \bigoplus\{v(\sigma) \mid \langle q', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q'', s'\rangle, \sigma \in \Delta_{post\text{-}2}\}$$

Finally, we use the fact that the flow algebra is affine:

$$\begin{aligned}
&\lambda_R^*(q' \overset{*}{\underset{\rho}{\xleftarrow{s}}} q) \\
&\sqsubseteq \bigoplus\{v(\sigma) \otimes v(\sigma') \mid \langle q', \epsilon\rangle \overset{\sigma'}{\Longrightarrow}{}^* \langle q'', s'\rangle, \langle q'', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, \gamma_\epsilon\rangle, \sigma, \sigma' \in \Delta_{post\text{-}2}\} \\
&\sqsubseteq \bigoplus\{v(\sigma) \otimes v(\sigma') \mid \langle q', \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle q, s\rangle, \sigma \in \Delta_{post\text{-}2}\}
\end{aligned}$$

□

B.3.8 Proof of Theorem 5.14

Consider an automaton $\mathcal{A}^{\mathcal{C}}_{post^*}$ constructed by the saturation procedure and the least solution λ to the set of its constraints $\mathcal{C}$. If the flow algebra is affine then for every path $\rho = q_f \overset{*}{\xleftarrow{s}} p$ where $q_f \in F$ we have that:

$$\lambda_R^*(q_f \overset{*}{\underset{\rho}{\xleftarrow{s}}} p) = \bigoplus\{v(\sigma) \mid \langle q_f, \epsilon\rangle \overset{\sigma}{\Longrightarrow}{}^* \langle p, s\rangle, \sigma \in \Delta^*_{post\text{-}2}\}$$

Proof. Follows directly from Theorem 5.5 and Lemma 5.13. □

Appendix C

Algorithms and examples from Chapter 8

C.1 Algorithms

Algorithms 2 and 3 present the graph-based algorithms for Pre^* and $Post^*$ computations. We skip the initialization phase (and handling of pop-rules in case of Pre^*), since it is quite simple and would only unnecessarily complicate the already non trivial code. Thus, both algorithms assume that the worklist has been initialized and present just the saturation stage of each procedure. Again, we do not consider constraint generation as it is quite simple and does not influence how the algorithms actually work.

```
input  : worklist, pds, wfa
output: final wfa representing A_pre*

while worklist is not empty do
    (pds-edge, target-state) ← Pop(worklist );
    (control-loc, stack-elem) ← Source(pds-edge );
    if ¬AddTransition(wfa, control-loc, stack-elem, target-state ) then
        continue with the next element;
    end
    foreach pds-edge2 ∈ Suspended(control-loc, stack-elem ) do
        Push(worklist, pds-edge2, target-state);
    end
    foreach pds-edge2 ∈ PdsInEdges(Source(pds-edge)) do
        stack-label ← Label(pds-edge2 );
        if stack-label = ϵ then
            Push(worklist, pds-edge2, target-state);
        else
            foreach target-state2 ∈ GetStates(target-state, stack-label ) do
                Push(worklist, pds-edge2, target-state2);
            end
            if IsInitial(target-state) then
                Insert(Suspended(target-state, stack-label), pds-edge2);
            end
        end
    end
end
```

Algorithm 2: Saturation stage of graph-based Pre^* algorithm.

```
input  : worklist, pds, wfa
output: final wfa representing A_post*

while worklist is not empty do
    (pds-edge, target-state) ← Pop(worklist);
    (control-loc, stack-elem) ← Target(pds-edge);
    stack-label ← Label(pds-edge);
    if stack-label = ε then
        if ¬AddTransition(wfa, control-loc, stack-elem, target-state) then
            continue with the next worklist element;
        end
        if stack-elem = ε then
            if ¬IsInitial(target-state) then
                Insert(EpsilonCache(target-state), control-loc);
            end
            foreach trans ∈ OutTrans(target-state) and pds-edge2 ∈
            PdsOutEdges((control-loc, Label(trans))) do
                Push(worklist, pds-edge2, Target(trans));
            end
        end
        foreach pds-edge2 ∈ PdsOutEdges(Target(pds-edge)) do
            Push(worklist, pds-edge2, target-state);
        end
    else
        intermediate ← GetIntermediate(control-loc, stack-elem);
        if ¬AddTransition(wfa, control-loc, stack-elem, intermediate) ∧
        ¬AddTransition(wfa, intermediate, stack-label, target-state) then
            continue with the next worklist element;
        end
        foreach predecessor ∈ EpsilonCache(intermediate) and pds-edge2 ∈
        PdsOutEdges((predecessor, stack-label)) do
            Push(worklist, pds-edge2, target-state);
        end
        foreach pds-edge2 ∈ PdsOutEdges(Target(pds-edge)) do
            Push(worklist, pds-edge2, intermediate);
        end
    end
end
```

Algorithm 3: Saturation stage of graph-based $Post^*$ algorithm.

C.2 Benchmark

We developed a simple generator of programs of various sizes for benchmarking purposes. Since we want to measure the performance of the Pre^* and $Post^*$ algorithms, we do not perform any actual analysis. Thus, to imitate an ordinary statement (i.e., not a procedure call) we simply use `skip` with the restriction that we do not optimize it away when constructing the pushdown graph. By passing an additional parameter to the generator we increase both the number of `foo` procedures and the size of the `main` procedure. The general shape of a generated program is as follows:

```
proc foo0
  skip;
  skip;
  a?(x);
  do
    :: x < 0 => call main
    :: 0 < x => call main
  od;
  skip;
  skip;
  skip
end
proc foo1
  skip;
  skip;
  a?(x);
  do
    :: x < 0 => call foo0
    :: 0 < x => call main
  od;
  skip;
  skip;
  skip
end
proc main
  skip;
  skip;
  skip;
  skip;
  skip;
  skip;
  a?(x);
```

```
  do
     :: x < 0 => call foo0
     :: 0 < x => call foo1
  od;
skip
end
```

Appendix D

Algorithms and examples for Chapter 9

D.1 Kleene star

D.1.1 Description

The algorithm for commutative case might require computing a Kleene star of a matrix and we have implemented two different algorithms to compute that. The first one is the well-known generalization of the Floyd-Warshall algorithm. For a given matrix $\mathbf{A}$ it computes its transitive closure, i.e., $\mathbf{A}^+$. By adding identity matrix $\mathbf{1}$ we get the $\mathbf{A}^*$. The complexity of the computation is immediate from the structure of the algorithm: $\Theta(n^3)$ [17].

The second one is a recursive divide-and-conquer algorithm [52, 51]. Since it is somewhat less known we will present it here briefly and show that it is also $\Theta(n^3)$ in the worst case. Let $\mathbf{M}$ be a square $m \times m$ matrix. We can divide it, for some $k \in \mathbb{N}$, into four submatrices as follows:

$$\mathbf{M} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

where $\mathbf{A}$ and $\mathbf{D}$ are square submatrices of dimensions $k \times k$ and $(m-k) \times (m-k)$ respectively. Let us introduce some intermediate matrices:

$$\mathbf{F} = \mathbf{A} + \mathbf{B}\mathbf{D}^*\mathbf{C}$$
$$\mathbf{G} = \mathbf{D} + \mathbf{C}\mathbf{A}^*\mathbf{B}$$

Then the $\mathbf{M}^*$ can be computed recursively as follows:

$$\mathbf{M}^* = \begin{bmatrix} \mathbf{F}^* & \mathbf{F}^*\mathbf{B}\mathbf{D}^* \\ \mathbf{G}^*\mathbf{C}\mathbf{A}^* & \mathbf{G}^* \end{bmatrix}$$

As we show below the procedure is also $\Theta(n^3)$ (the main factor here is the matrix multiplications performed at each step of the recursion).

Since both procedures have the same asymptotic complexity one might ask whether it is useful to implement both of them. However, the semiring expressions that arise in both computations are often quite different and we have found that for some abstractions (e.g., semilinear sets discussed in Section 9.3) they can significantly influence the performance. However, we have not found one of the algorithms to always provide better performance across different semirings.

D.1.2 Runtime analysis

Here we will consider in a bit more detail the runtime analysis of the divide-and-conquer algorithm. We will use the master theorem [17], which allows to solve some of the recurrences of the form:

$$R(n) = aR\left(\frac{n}{b}\right) + f(n)$$

In our case we try to minimize the number of recursive calls by splitting the matrix roughly in half, i.e., $k = \lfloor \frac{n}{2} \rfloor$. We have four recursive calls, each on the matrix of the dimension $\lfloor \frac{n}{2} \rfloor$ as well as eight matrix multiplications. However, with some common subexpression elimination it is easy to have just six matrix multiplications and so our recurrence relation is:

$$R(n) = 4R\left(\frac{n}{2}\right) + 6\left(\frac{n}{2}\right)^3$$

In order to apply one of the cases of the master theorem we need to show that there exists an ϵ such that $f(n) \in \Omega(n^{\log_b a+\epsilon})$. Now we clearly have that $\log_2 4 = 2$ and $6\left(\frac{n}{2}\right)^3 \in \Theta(n^3)$. We also need to show that $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$. It is easy to see that there exists such c:

$$4\left(\frac{n}{4}\right)^3 = \frac{1}{16}n^3 < c * 6\left(\frac{n}{2}\right)^3 = \frac{3}{4}cn^3$$

thus, it follows from the master theorem that the whole procedure is $\Theta(n^3)$.

Finally, it is useful to consider the cost of evaluating such a matrix. Since the procedure can create expressions that (due to sharing of subexpressions) are linear in the number of variables $|\mathcal{X}|$ and there are $|\mathcal{X}|^2$ number of such expressions in the matrix, we have that the cost of evaluating it is $\mathcal{O}(|\mathcal{X}|^3)$. However, this bound is quite pessimistic, e.g., equation systems arising from program analysis problems are rarely very dense and many of the cells of the adjacency matrix will have value $\bar{0}$. Also note that it is essential that we take advantage of the sharing of nodes and cache already evaluated subexpressions (thus avoiding the exponential complexity). Finally, we also assume that the insertion and lookup in a hash table is $\mathcal{O}(1)$.

D.2 Benchmarks

The first example is presented below.

```
proc main
  a?(x);
  do
     :: x = 1 => b!(); c!()
     :: x = 2 => c!(); d!()
     :: x = 3 => d!(); b!()
  od;
  a?(x);
  if
     :: x < 0     => call main; call main
     :: !(x < 0) => skip
  fi;
  a!()
end
```

The second example's main difference with the first one is the fact that during constraint solving there will be some constraints that can be abstracted by dividing the elements by their GCD.

```
proc main
  a?(x);
  do
     :: x = 1 => b!(); b!(); b!()
     :: x = 2 => b!(); b!(); b!(); b!()
     :: x = 3 => d!(); d!(); d!()
```

```
  od;
  a?(x);
  if
    :: x < 0     => call main; call main
    :: !(x < 0) => skip
  fi;
  a!()
end
```

Finally, in the third example we add an additional call to `main` that increases the degree of the equation system.

```
proc main
  a?(x);
  do
    :: x = 1 => b!(); c!()
    :: x = 2 => c!(); d!()
    :: x = 3 => d!(); b!()
  od;
  a?(x);
  if
    :: x < 0     => call main; call main; call main
    :: !(x < 0) => skip
  fi;
  a!()
end
```

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Pearson/Addison Wesley, Boston, MA, USA, second edition, 2007.

[2] Mohamed Faouzi Atig, Ahmed Bouajjani, and Tayssir Touili. On the reachability analysis of acyclic networks of pushdown systems. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 356–371. Springer, 2008.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008.

[4] Gogul Balakrishnan, Thomas W. Reps, Nicholas Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, Suan Hsi Yong, Chi-Hua Chen, and Tim Teitelbaum. Model checking x86 executables with codesurfer/x86 and wpds++. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2005.

[5] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *IFM*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2000.

[7] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In John Launchbury and John C. Mitchell, editors, *POPL*, pages 1–3. ACM, 2002.

[8] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2012.

[9] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[10] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 62–73. ACM, 2003.

[11] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. *Int. J. Found. Comput. Sci.*, 14(4):551–582, 2003.

[12] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[13] Johannes Bubenzer. Minimization of acyclic DFAs. In Jan Holub and Jan Žďárek, editors, *Proceeding of the Prague Stringology Conference 2011*, pages 132–146, Czech Technical University in Prague, Czech Republic, 2011.

[14] Sagar Chaki, Edmund M. Clarke, Nicholas Kidd, Thomas W. Reps, and Tayssir Touili. Verifying concurrent message-passing c programs with recursive calls. In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2006.

[15] Yvonne Coady, Gregor Kiczales, Michael J. Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ESEC / SIGSOFT FSE*, pages 88–98, 2001.

[16] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[18] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In E.J. Neuhold, editor, *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, pages 237–277. North-Holland, 1977.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *POPL*, pages 238–252. ACM, 1977.

[20] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen, editors, *POPL*, pages 269–282. ACM Press, 1979.

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35. ACM Press, 1989.

[22] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[23] Manfred Droste and Werner Kuich. Semirings and formal power series. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science. An EATCS Series, pages 3–28. Springer Berlin Heidelberg, 2009.

[24] The Erlang programming language. http://www.erlang.org/.

[25] Javier Esparza. *Automata theory: An algorithmic approach (Lecture notes)*. July 2012.

[26] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2000.

[27] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *J. ACM*, 57(6):33, 2010.

[28] Javier Esparza and Michael Luttenberger. Solving fixed-point equations by derivation tree analysis. In Andrea Corradini, Bartek Klin, and Corina Cîrstea, editors, *CALCO*, volume 6859 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2011.

[29] Piotr Filipiuk, Hanne Riis Nielson, and Flemming Nielson. Explicit versus symbolic algorithms for solving ALFP constraints. *Electr. Notes Theor. Comput. Sci.*, 267(2):15–28, 2010.

[30] Piotr Filipiuk, Michal Terepeta, Hanne Riis Nielson, and Flemming Nielson. Galois connections for flow algebras. In Roberto Bruni and Jürgen Dingel, editors, *FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2011.

[31] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *ML*, pages 12–19. ACM, 2006.

[32] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In Chris Hankin and Dave Schmidt, editors, *POPL*, pages 193–205. ACM, 2001.

[33] Seymour Ginsburg and Edwin Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16(2):285–296, 1966.

[34] The Go programming language. http://golang.org/.

[35] Nicolas Halbwachs and Lenore D. Zuck, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*. Springer, 2005.

[36] Chris Hankin, Flemming Nielson, Hanne Riis Nielson, and Fan Yang. Advice for coordination. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2008.

[37] The Haskell programming language. http://www.haskell.org.

[38] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 232–244. ACM, 2004.

[39] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[40] C. A. R. Hoare. How did software get so reliable without proof? In Marie-Claude Gaudel and Jim Woodcock, editors, *FME*, volume 1051 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996.

[41] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation (3rd Edition)*. Addison-Wesley, 2007.

[42] Somesh Jha and Thomas W. Reps. Analysis of spki/sdsi certificates using model checking. In *CSFW*, pages 129–. IEEE Computer Society, 2002.

[43] Somesh Jha and Thomas W. Reps. Model checking spki/sdsi. *Journal of Computer Security*, 12(3-4):317–353, 2004.

[44] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[45] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[46] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[47] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[48] Nicholas Kidd, Akash Lal, and Thomas W. Reps. Wali: The weighted automaton library, December 2007.

[49] Gary A. Kildall. A unified approach to global program optimization. In *POPL*, pages 194–206, 1973.

[50] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In Uwe Kastens and Peter Pfahler, editors, *CC*, volume 641 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 1992.

[51] Lucja Kot and Dexter Kozen. Kleene algebra and bytecode verification. *Electr. Notes Theor. Comput. Sci.*, 141(1):221–236, 2005.

[52] Dexter Kozen. *Automata and computability*. Undergraduate texts in computer science. Springer, 1997.

[53] Akash Lal. *Interprocedural Analysis and the Verification of Concurrent Programs*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, August 2009.

[54] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In Ramakrishnan and Rehof [62], pages 282–298.

[55] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas W. Reps. Interprocedural analysis of concurrent programs under a context bound. In Ramakrishnan and Rehof [62], pages 282–298.

[56] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[57] Michael Luttenberger. *Solving Systems of Polynomial Equations: A Generalization of Newton's Method*. PhD thesis, Technische Universität München, 2010.

[58] Michael Luttenberger and Maximilian Schlund. An extension of parikh's theorem beyond idempotence. *CoRR*, abs/1112.2864, 2011.

[59] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis (2. corr. print)*. Springer, 2005.

[60] Flemming Nielson, Helmut Seidl, and Hanne Riis Nielson. A succinct solver for ALFP. *Nord. J. Comput.*, 9(4):335–372, 2002.

[61] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Halbwachs and Zuck [35], pages 93–107.

[62] C. R. Ramakrishnan and Jakob Rehof, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*. Springer, 2008.

[63] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Program. Lang. Syst.*, 22(2):416–430, 2000.

[64] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *POPL*, pages 49–61. ACM Press, 1995.

[65] Thomas W. Reps, Akash Lal, and Nicholas Kidd. Program analysis using weighted pushdown systems. In Vikraman Arvind and Sanjiva Prasad, editors, *FSTTCS*, volume 4855 of *Lecture Notes in Computer Science*, pages 23–51. Springer, 2007.

[66] Thomas W. Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In Radhia Cousot, editor, *SAS*, volume 2694 of *Lecture Notes in Computer Science*, pages 189–213. Springer, 2003.

[67] Thomas W. Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.

[68] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):pp. 358–366, 1953.

[69] Barry K. Rosen. Monoids for rapid data flow analysis. *SIAM J. Comput.*, 9(1):159–196, 1980.

[70] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1&2):131–170, 1996.

[71] Maximilian Schlund and Michal Terepeta. Newton on semirings. Technical report, 2013.

[72] Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technical University Munich, 2002.

[73] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice Hall Professional Technical Reference, Englewood Cliffs, NJ, 1981.

[74] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in aop with aspectc++. In Hamido Fujita and Mohamed Mejri, editors, *SoMeT*, volume 129 of *Frontiers in Artificial Intelligence and Applications*, pages 33–53. IOS Press, 2005.

[75] Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. jmoped: A java bytecode checker based on moped. In Halbwachs and Zuck [35], pages 541–545.

[76] Michal Terepeta, Hanne Riis Nielson, and Flemming Nielson. Recursive advice for coordination. In Marjan Sirjani, editor, *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2012.

[77] Michal Terepeta, Hanne Riis Nielson, and Flemming Nielson. Pushdown systems for monotone frameworks. *CoRR*, abs/1307.4585, 2013.

[78] Bruce W. Watson. A new algorithm for the construction of minimal acyclic dfas. *Sci. Comput. Program.*, 48(2-3):81–97, 2003.

[79] Bruce William Watson. *Constructing Minimal Acyclic Deterministic Finite Automata*. PhD thesis, University of Pretoria, 2010.

[80] Fan Yang. *Aspects with Program Analysis for Security Policies*. PhD thesis, Technical University of Denmark, 2010.