UNIVERSITY OF LEICESTER

DEPARTMENT OF INFORMATICS

THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Recurrent Sets for Non-Termination and Safety of Programs

Alexey Bakhirkin

September 2016

Recurrent Sets for Non-Termination and Safety of Programs

Alexey Bakhirkin

Termination and non-termination are a pair of fundamental program properties. Arguably, the majority of code is required to terminate, e.g., dispatch routines of drivers or other event-driven code, GPU programs, etc – and the existence of non-terminating executions is a serious bug. Such a bug may manifest by freezing a device or an entire system, or by causing a multi-region cloud service disruption. Thus, proving termination is an interesting problem in the process of establishing correctness, and proving non-termination is a complementary problem that is interesting for debugging.

This work considers a sub-problem of proving non-termination – the problem of finding recurrent sets. A recurrent set is a way to compactly represent the set of non-terminating executions of a program and is a set of states from which an execution of the program cannot or may not escape (there exist multiple definitions that differ in modalities). A recurrent set acts as a part of a non-termination proof. If we find a non-empty recurrent set and are able to show its reachability from an initial state – then we prove the existence of a non-terminating execution.

Most part of this work is devoted to automated static analyses that find recurrent sets in imperative programs. We follow the general framework of abstract interpretation and go all the way from trace semantics of programs to practical analyses that compute abstract representations of recurrent sets. In particular, we present two novel analyses. The first one is based on abstract pre-condition computation (backward analysis) and trace partitioning and focuses on numeric programs (but with some modifications it may be applicable to non-numeric ones). In popular benchmarks, it performs comparably to state-of-the-art tools. The second analysis is based on abstract post-condition computation (forward analysis) and is readily applicable to non-numeric (e.g., heap-manipulating) programs, which we demonstrate by tackling examples from the domain of shape analysis with 3-valued logic.

As it turns out, recurrent sets can be used in establishing other properties as well. For example, recurrent sets are used in CTL model checking of programs. And as part of this work, we were able to apply recurrent sets in the process of establishing sufficient pre-conditions for safety.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work is about program analysis. Though, this explanation is too abstract, and we need to refine it before we can describe the actual contents of our research.

First, we are going to perform *static analysis* meaning that we will identify some runtime *properties* of programs without actually running them. Second, we will analyse imperative programs. As an imperative program runs, it goes through some sequence of states, which we call an *execution*. A single program may have multiple possible executions: the program may start in one of the many possible initial states, and some statements in the program may have multiple possible outcomes. In this setting, a *property* of a program is a mathematical object, containing some information about the set of *all* executions of the program: e.g., whether the program may ever reach an erroneous state, whether the program always (sometimes, never) terminates, etc. This means that we will *not* perform *testing* and observe a finite number of executions of the program. Instead, we will build a mathematical *model* of a program and from the properties of the model will derive properties of the set of its executions. Finally, our analysis will be *automatic*, meaning that we will produce an *algorithm* that can identify some properties of programs without interacting with a human. Thus, this work is about *automated static analysis of imperative programs*.

This may seem as a problem that is too hard to tackle[1], but in practice, this is not always the case. It turns out that programs, for which static analysis is important, are often feasible to analyse. For example, many fragments of critical systems code are what

[1] And indeed, program analysis is a hard problem. When modelling the state of a program as a bit vector, analysis becomes NP-complete. Analysis of models with infinite state-space often becomes undecidable.

is called *control intensive*. That is, their behaviour is mostly encoded in the program text and to a lesser extent depends on input data. Also, control-intensive programs are often numeric (i.e., important variables are numbers) and often even linear (i.e., important statements perform affine transformations of the variables). This allows to use in the analysis all the numerous achievements in linear programming[2]. Additionally, as academic researchers, we often allow ourselves to avoid dealing with quirks and peculiarities of a particular programming language and restrict ourselves to models of programs that maintain only the essential aspects of the behaviour.

We now proceed to the discussion of exactly which properties we are interested in and what will be our main tool in identifying them.

1.1 Non-Termination and Termination

In this work, we are interested in a specific property of programs, which can be informally described as, *"When does a program not terminate?"*

Termination and non-termination are a pair of fundamental properties of computer programs. Arguably, the majority of code is required to terminate, e.g., dispatch routines of drivers or other event-driven code, GPU programs, etc – and the existence of non-terminating executions is a serious bug. Such a bug may manifest by freezing a device or an entire system or by causing a multi-region cloud service disruption[3]. Thus, proving termination becomes an interesting problem, as part of the process of establishing correctness of a program. At the same time, this problem (i.e., halting problem) is in general undecidable. That is, if an automatic technique can soundly prove ter-

[2] Some numeric techniques, to be sound, require that all numeric variables in a program can take arbitrary integer or rational values. In particular, that the value of an integer variable does not "wrap around" or saturate when increasing or decreasing past a certain point. While it is in general not consistent with the behaviour of machine integer and floating-point numbers (which can only take the values from certain finite subsets of $\mathbb{Z}$ and $\mathbb{Q}$), this requirement is actually quite benign. An analysis can always check whether in a given program, some variable may reach a critical value and if so, declare the results of the analysis as non-conclusive. Another point is that in some programming languages, the effect of an overflow of a signed integer variable is considered undefined, and it is just not possible to soundly predict the behaviour of a program in this case. For these reasons, academic researchers often work with models of programs where there is no bound on the values of numeric variables. In this work, the techniques will not rely on the absence of overflows and will allow to encode different semantics of numbers. On the other hand, some examples will assume that integer variables can take arbitrarily large positive or negative values, but this is solely to simplify the presentation.

[3] `http://azure.microsoft.com/blog/2014/11/19/update-on-azure-storage-service-interruption`, last accessed in May 2016

```
1  days ← a number ≥ 0
2  year ← 1980
3  while (days > 365) {
4    if (leap(year)) {
5      if (days > 366) {
6        days ← days − 366;
7        year ← year + 1;
8      }
9    } else {
10     days ← days − 365;
11     year ← year + 1;
12   }
13 }
```

Figure 1.1: Demonstration of a real-life non-termination bug.

```
1  days ← a number ≥ 0
2  year ← 1980
3  while (days > 365) {
4    if (leap(year)) {
5      if (days > 366) {
6        days ← days − 366;
7        year ← year + 1;
8      } else if (days = 366)
9        days ← days − 366;
10   } else {
11     days ← days − 365;
12     year ← year + 1;
13   }
14 }
```

Figure 1.2: A possible fix for the non-termination bug.

mination of some programs[4], this technique is incomplete. If it fails to prove that a program terminates, this does not mean that the program has non-terminating executions. This way, proving non-termination becomes an interesting complementary problem, which arises in the process of finding bugs in programs (i.e., debugging in the sense of [Bou93a]). In general, we will use the phrases, *"to prove termination"* and *"to prove non-termination"* in the following sense. By proving termination we mean finding the initial states from which a program terminates or showing that the program terminates from all initial states. Conversely, by proving non-termination we mean showing that from all or from some specific initial states at least one non-terminating execution can originate. Let us illustrate this all with an example.

Example 1.1. Fig. 1.1 demonstrates a non-termination bug that actually occurred in the software of Microsoft Zune players in the end of 2008. This is a simplified fragment of a procedure which calculates current date (in the example – only the current year)

[4]Undecidability of halting problem does not mean that we cannot prove termination or non-termination of any program. It means that there is no algorithm that for every program will give a definite answer on whether or not the program terminates. Still, there exist algorithms that can prove termination (or non-termination) of some programs.

based on the number of days that have passed since 1 January 1980. In this fragment, the loop repeatedly subtracts 365 or 366 from the current number of days depending on whether the current year is leap and then increases the year by 1. Due to a logical error, if the current year is leap and the current number of days is exactly 366, a loop iteration does not change the variables, and the program goes into an infinite loop. This happened for many devices on 31 December 2008 causing them to freeze with the only option for an affected user being to wait until the battery of the device completely discharges and then turn on the device the day after [5].

Overall, non-termination bugs, although they often cause a great impact especially when they occur in systems code, seem to usually have simple explanations. Thus, there is hope that automated analyses (unlike humans) will be good at finding such bugs. How would an automated *termination* prover help in this case? The general structure of a termination proof was offered by Alan Turing back in the end of 1940s [Tur49]. To show that a program terminates, one (a person or an algorithm) would need to construct a structure-preserving map (usually called a *ranking function*) from the state-space of the program (together with the transition relation of the program) to some set equipped with a well-founded relation. One way to think of a ranking function is that for every state of the program, it should give an upper bound on the number of computation steps until termination. For the program in Fig. 1.1 there is no such ranking function, and a non-termination prover will report that it cannot find one. This will be a hint to the programmer that something *might* be wrong (as mentioned above, non-termination provers are incomplete, and failure to find a ranking function in does not in the general case mean that there is no ranking function at all). On the other hand, if we apply an automated *non-termination* prover to the program in Fig. 1.1, we will get one of the two possible outcomes:

(i) the non-termination prover will be able to show the existence of states from which the program does not terminate and expose this fact to the programmer thus showing what exactly the non-termination bug is;

(ii) the non-termination prover will fail to find any non-terminating execution which (due to incompleteness) leaves us with a non-conclusive result. Failure to find a non-terminating execution does not mean there does not exist one.

[5] `http://www.zuneboards.com/forums/showthread.php?t=38143`, last accessed in May 2016.

If we get rid of the non-termination bug, e.g., as in Fig. 1.2, a sound (and they are usually expected to be sound) non-termination prover will definitely not find any non-terminating executions. A termination prover may (or may not) find a ranking function and demonstrate that with every loop iteration the current number of days always decreses (but never goes below 0).

This is a common situation for incomplete automated analyses. An analysis will infer some property of a program that is itself a mathematical object: a ranking function, some representation of non-terminating executions, etc. Then we can try to use this object to answer a binary question, e.g., *"Does this program always terminate?"*. Sometimes, we will be able to obtain a definite answer, and sometimes we will not. For example, if we are able to find a ranking function, the answer is a definite *Yes*. On the other hand, if a termination prover did not find a ranking function, the answer is indefinite, i.e., *Maybe*. Similarly, if a non-termination prover is able to find at least one non-terminating execution, the answer is a definite *No*. If the non-termination prover fails to find one, the answer is still *Maybe*.

The success in showing termination of imperative programs is a relatively recent achievement[6]. The general structure of a termination proof is due to Turing, but the algorithm to infer ranking functions for practical numeric programs was offered in 2004 by Andreas Podelski and Andrey Rybalchenko [PR04a; PR04b] (using other long-known results: Ramsey theorem [Ram30] and Farkas' lemma [Sch99]) and later implemented together with Byron Cook [CPR06], with the goal of showing termination of dispatch routines of device drivers. After initial success with proving termination of imperative programs, the results for proving non-termination were to follow [Gup+08; VR08].

1.2 Recurrent Sets

One way to compactly represent the set of non-terminating executions of a program is via a *recurrent set*, which is used by a number of modern analyses [Che+14; Coo+14;

[6] We come from the program analysis background and thus our approach is to find a way to apply standard program analysis methods (in particular, our main technique will be abstract interpretation) to termination and non-termination analysis. On the other hand, we have to mention that termination has been extensively studied in other contexts as well; in particular – in the context of term rewriting systems. Advances in that field have been successfully applied to proving termination of programs and implemented in the tool AProVE [Gie+14].

```
1  days ← a number ≥ 0
2  year ← 1980
3  while (days > 365) {
4    if (*) {
5      if (days > 366) {
6        days ← days − 366;
7        year ← year + 1;
8      }
9    } else {
10     days ← days − 365;
11     year ← year + 1;
12   }
13 }
```

Figure 1.3: An abstraction of the program in Fig. 1.1.

Lar+14]. There exist multiple different definitions, but in general this is a set of states from which an execution of the program cannot or may not escape.

In the context of proving non-termination, a recurrent set acts as a part of a non-termination proof. Finding a non-empty recurrent set and showing reachability of some state in it from some initial state – amounts to proving the existence of a non-terminating execution. If we find an empty recurrent set or cannot show its reachability then we obtain non-conclusive results: it could either be that the program indeed does not have non-terminating executions, or that due to incompleteness, we overlooked them during the analysis.

As it turns out, recurrent sets can be used in establishing other properties as well. For example, we were able to apply recurrent sets in the process of establishing sufficient pre-conditions for safety. This is the topic of Chapter 6. There is also research that to our knowledge uses recurrent sets in CTL model checking of programs [CKP15].

Recurrent sets are the main focus of this work. Thus, we can finally say that this work is about *automated static analysis that finds recurrent sets in imperative programs.*

1.3 Abstraction

The main technique that allows us to reason about infinite or very large mathematical objects and thus solve in some cases the problems that are in general undecidable is abstraction. In program analysis, abstraction takes two main forms.

(i) One form is abstraction of statements which means that we replace some statements of the program with their approximate versions with the intention to simplify the computation of the interesting property. For example, recall the program in Fig. 1.1. The condition *leap*(*year*) is actually hard to work with as it cannot be represented precisely in many abstract domains[7]. In particular, divisibility by a constant can be represented in the domain of linear congruences, but indivisibility cannot. In an analysis, we may want to simplify this condition, and in the extreme case, we can just replace the `if`-statement by a non-deterministic branching as shown in Fig. 1.3. Replacing procedure calls with non-deterministic effects is also common when not all the source code for a program is available. Abstraction of statements changes the behaviour of programs. Observe that in the concrete program in Fig. 1.1, the states where the year is leap and the number of days is 366 have only non-terminating executions originating in them. In the abstracted program in Fig. 1.3, all states where the number of days is 366 have both terminating and non-terminating executions originating from them. In abstract interpretation, the program is usually not modified explicitly, but the effects of individual statements are abstracted during the approximate computation.

(ii) The second form is abstraction of properties. Instead of trying to compute an exact property of a program (ranking function, recurrent set, etc), we will usually be computing an approximate property of a certain form. The form of the approximate property is often called an abstract domain. For example, in a numeric program (where all the interesting variables are numbers) we may be looking for a recurrent set in the form of a conjunction of linear inequalities (or for a ranking function in piecewise-linear form). In this case, for the program in Fig. 1.1, we may discover a number of singleton recurrent sets, e.g., $year = 1980 \wedge days = 366$, $year = 1984 \wedge days = 366$, $year = 1988 \wedge days = 366$, etc; but we may not be able to

[7] Recall that a year is leap either when it is divisible by 4, but not 100; or when it is divisible by 400.

find the largest recurrent set as it cannot be represented as a conjunction of linear inequalities. This form of abstraction usually comes together with abstraction of statements, which are abstracted to be transformers in the abstract domain.

The common requirement when computing an approximate property is for it to be sound, i.e. to preserve the ability to (sometimes) give definite answers to binary questions. For example, we will usually want our analysis to find an *under-approximation* of the actual recurrent set of a program (a set that is included in the actual recurrent set). The effect is that if we find a non-empty approximate recurrent set, the actual recurrent set is definitely non-empty.

1.4 Contents and Contribution

Thus, in this work we study the techniques for finding recurrent sets and their applications. In Chapter 2, we prepare theoretical background and give formal meaning to everything that we mentioned in the introduction. We define a formal notion of a program and its execution; we demonstrate a systematic way to apply abstraction when analysing programs (abstract interpretation); we give two notions of recurrent sets and show how they relate the set of a non-terminating executions of a program.

In Chapters 3–5 we address practical problems of computing recurrent sets for programs (in an under-approximate way). In particular, in Chapter 3, we focus on numeric programs and develop a technique to find so called existential recurrent sets (i.e., sets of states that might not be escaped, depending on which non-deterministic choices a program takes). The technique is based on backward analysis (computation of predecessors of states in the program) and trace partitioning (a technique to perform the analysis separately for different paths through the program). An additional contribution of Chapter 3 is that (based on the material of Chapter 2) it formally describes trace partitioning for backward analysis, which to our knowledge has not been done before.

In Chapter 4, we address the issue of expensiveness of backward analysis and offer an analysis for so called universal recurrent sets (i.e., sets of states that cannot be escaped, regardless of non-deterministic choices) that is based on post-condition computation. The analysis builds an abstract reachability graph of a program, in a way similar to some existing program model-checkers, and analyses it in a novel way. We

show that this analysis can be applied to heap-manipulating programs, for which to our knowledge no procedure for finding recurrent sets was known before. In particular, we implemented support for shape analysis with 3-valued logic [SRW02].

In Chapter 5, we report on our experimental results with two sets of benchmarks. We show that a prototype implementation of the algorithm of Chapter 3 demonstrates precision that is on the same level with state-of-the-art tools.

In Chapter 6, we demonstrate how the notion of recurrent set can be used in an analysis for safety and propose a novel approach for computing weakest liberal safe preconditions of programs. The approach will compute the set of safe state as a least fixed-point above a recurrent set, and ensures soundness by subtracting from the potentially safe states an over-approximation of the unsafe states.

1.5 Timeline and Motivation

This section describes some of the choices that we took in this work as some of them arguably are unconventional. For example, we use abstract interpretation to analyse even numeric programs, instead of using some form of abstraction refinement model checking procedure (e.g., Impact [McM06]). Also, we use shape analysis with 3-valued logic [SRW02] to analyse heap-manipulating programs, instead of, e.g., separation logic [Rey02]. This section is meant to explain some of the choices that we take in this work.

In 2012, we chose shape analysis with 3-valued logic as our initial research direction, and the goal was to see whether we can come up with a novel analysis procedure or with a novel way to represent program heaps, that would be based on 3-valued logic. This initial research effort resulted in a paper on backward analysis for sufficient preconditions that was presented and published in SAS[8] in 2014 [BBP14]. It is described in detail in Chapter 6. The SAS'2014 paper influenced further research in two ways.

First, it made use of abstract interpretation, as it is the main technique that is available for shape analysis with 3-valued logic. In particular, abstract interpretation allows us to develop analyses that are *parameterized* with an abstract domain (the way in which we represent abstract), i.e., the same analysis can be applied to different kinds

[8] Static Analysis Symposium.

of programs (numeric, heap-manipulating, etc.). Usually, there are relatively few requirements for a domain to be usable. Typically, we need a way to compute certain state transformers (pre-condition and/or post-condition), and a way to ensure that the analysis of a program eventually terminates. Analyses based on abstract interpretation can be instantiated with different domains: 3-valued logic or separation logic for heap-manipulating programs, various numeric domains, etc.

Second, it drew our attention to the notion of recurrent set and the problem of proving non-termination of programs. In the SAS'2014 paper, finding a recurrent set was a sub-problem of finding a sufficient pre-condition for safety, but soon we realized that it is an interesting problem by itself. From that point on, our research was focusing on finding recurrent sets using abstract interpretation.

One of the takeaways from the SAS'2014 paper was that backward analysis (that based on computing pre-conditions) with 3-valued logic is difficult to implement and computationally expensive. Thus, in our next research effort (which was focused on finding recurrent sets), we decided to try and use forward analysis (that based on computing post-conditions). Shape analysis was no longer the focus of the research, but we still wished to be able to use complicated abstract domains (e.g., 3-valued logic for heap-manipulating programs) and thus we continued to use abstract interpretation. This research effort resulted in a paper that was presented and published in SAS in 2015. It is described in detail in Chapter 4.

In the SAS'2015 paper, we made an observation that for many programs, the non-terminating executions take a specific path (or paths) through the program. In our next research effort, we wished to exploit this fact and develop an analysis that would be able to infer which paths the non-terminating executions take. To do that, we followed an already familiar approach of abstract interpretation: we made the set of program paths into the concrete domain and came up with an abstract path domain: a way to finitely represent interesting sets of paths. This allowed us to develop an analysis that at the same time infers the form of non-terminating paths and the pre-condition for the program to take them (this technique is called trace partitioning and was first described by Laurent Mauborgne and Xavier Rival [RM07]). As a result, we ended up using abstract interpretation to analyse numeric programs, even though arguably, other analysis techniques currently dominate this area. This research effort resulted in a pa-

per that was presented and published in TACAS[9] in 2016. It is described in detail in Chapter 3.

1.6 Publications

This work is based on the following publications. All the publications are my own works, performed under supervision of Josh Berdine and Nir Piterman.

[BP16] Alexey Bakhirkin and Nir Piterman. "Finding Recurrent Sets with Backward Analysis and Trace Partitioning". In: *Tools and Algorithms for the Construction and Analysis of System (TACAS)*. ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 17–35 – Chapter 3, parts of Chapters 2 and 5.

[BBP15] Alexey Bakhirkin, Josh Berdine, and Nir Piterman. "A Forward Analysis for Recurrent Sets". In: *Static Analysis Symposium (SAS)*. ed. by Sandrine Blazy and Thomas Jensen. Vol. 9291. Lecture Notes in Computer Science. Springer, 2015, pp. 293–311 – Chapter 4, parts of Chapters 2 and 5.

[BBP14] Alexey Bakhirkin, Josh Berdine, and Nir Piterman. "Backward Analysis via over-Approximate Abstraction and under-Approximate Subtraction". In: *Static Analysis Symposium (SAS)*. ed. by Markus Müller-Olm and Helmut Seidl. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 34–50 – Chapter 6.

[9] International Conference on Tools and Algorithms for the Construction and Analysis of Systems, part of European Joint Conferences on Theory and Practice of Software.

Chapter 2

Background

The goal of this chapter is to prepare theoretical background for Chapters 3–6 (which will be more practical in nature).

In Section 2.1 we introduce basic mathematical notation.

In Section 2.2 we introduce the theoretical notion of a program and its behaviour. The discussion of programs continues in Section 2.5, where we introduce a restricted notion of structured program.

In Section 2.3 we briefly describe abstract interpretation, which is a well known systematic approach to analysing programs. In particular, we show how we can use abstract interpretation to analyse the behaviour of programs.

Finally, Section 2.4 introduces the analysis for non-termination on a theoretical level and describes the connection between non-termination analysis and the notion of a recurrent set, which is a set of states that a program cannot or may not escape. Later, Chapters 3–4 will address the practical side of finding recurrent sets.

2.1 Basic Notation

We write 1 and 0 to mean logical truth and falsity respectively. We use Kleene's 3-valued logic [Kle87] to represent truth values of formulas in sets. The logic uses three values $\mathcal{K} = \{0, 1, {}^1/_2\}$ meaning *false*, *true*, and *maybe* respectively. The meaning of standard logical operators in Kleene logic is given in Fig. 2.1. $\mathcal{K}$ is arranged in partial *information order* $\sqsubseteq_{\mathcal{K}}$, s.t. 0 and 1 are incomparable, $0 \sqsubseteq_{\mathcal{K}} {}^1/_2$, and $1 \sqsubseteq_{\mathcal{K}} {}^1/_2$. For $k_1, k_2 \in \mathcal{K}$,

$\neg$	
$\mathbf{0}$	1
$\mathbf{1}$	0
$\mathbf{1/2}$	$1/2$

(a) Negation.

$\wedge$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1/2}$
$\mathbf{0}$	0	0	0
$\mathbf{1}$	0	1	$1/2$
$\mathbf{1/2}$	0	$1/2$	$1/2$

(b) Conjunction.

$\vee$	$\mathbf{0}$	$\mathbf{1}$	$\mathbf{1/2}$
$\mathbf{0}$	0	1	$1/2$
$\mathbf{1}$	1	1	1
$\mathbf{1/2}$	$1/2$	1	$1/2$

(c) Disjunction.

Figure 2.1: Logical operators in 3-valued logic.

the least upper bound $\sqcup_{\mathscr{K}}$ is defined as:

$$k_1 \sqcup_{\mathscr{K}} k_2 = \begin{cases} k_1, \text{ if } k_1 = k_2 \\ 1/2, \text{ otherwise} \end{cases}$$

For a set S, we write Δ_S to mean the diagonal relation on S:

$$\Delta_S = \{(s, s) \mid s \in S\}$$

For a relation $T \subseteq S \times S$, we sometimes write $T(s, s')$ to mean $(s, s') \in T$. We use $\circ$ for *right* composition of relations:

$$T_2 \circ T_1 = \{(s, s'') \mid \exists s'.\ (s, s') \in T_1 \wedge (s', s'') \in T_2\}$$

We define k-th power T^k of a relation $T \subseteq S \times S$, s.t.

$$T^k = \begin{cases} \Delta_S, \text{ if } k = 0 \\ T^{k-1} \circ T, \text{ for } k \geq 1 \end{cases}$$

For a set S, we write $\mathscr{P}(S)$ to mean the powerset of S, i.e., the set $\{X \mid X \subseteq S\}$ of all subsets of S.

Let a set $\mathscr{L}$ be partially ordered by $\sqsubseteq$ (i.e., $\sqsubseteq$ is a reflexive, transitive, and antisymmetric relation on $\mathscr{L}$) and a set $\mathscr{L}'$ be partially ordered by $\sqsubseteq'$. We say that a function $F\colon \mathscr{L} \to \mathscr{L}'$ is monotone iff for every $l_1, l_2 \in \mathscr{L}$, $l_1 \sqsubseteq l_2 \Rightarrow F(l_1) \sqsubseteq' F(l_2)$.

We say that $\mathscr{L}$ is a complete lattice $\mathscr{L}\langle \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ when

(i) $\mathscr{L}$ is partially ordered by $\sqsubseteq$;

(ii) for every subset $X \subseteq \mathscr{L}$, there exists the least upper bound $\bigsqcup X \in \mathscr{L}$ and the great-

est lower bound $\bigsqcap X \in \mathscr{L}$; for a pair of elements $X = \{x_1, x_2\}$, we write $x_1 \sqcup x_2$ and $x_1 \sqcap x_2$ respectively;

A complete lattice has the least (w.r.t. $\sqsubseteq$) element that we denote by $\bot$ and the greatest element that we denote by $\top$.

For a monotone function on a complete lattice $F : \mathscr{L} \to \mathscr{L}$, we denote the least fixed point of F (the least element $l \in \mathscr{L}$, s.t. $F(l) = l$; it necessarily exists due to Knaster-Tarski theorem [Tar55]) by $\mathrm{lfp}_{\sqsubseteq} F$, and the greatest fixed point – by $\mathrm{gfp}_{\sqsubseteq} F$.

Occasionally, we use lambda-notation to write functions. For example, $\lambda x.x^2$ denotes a function that maps every number to its square. Alternatively, we use placeholder notation writing, e.g., $f(\cdot) + 1$ to mean $\lambda x. f(x) + 1$.

2.2 Programs and Executions

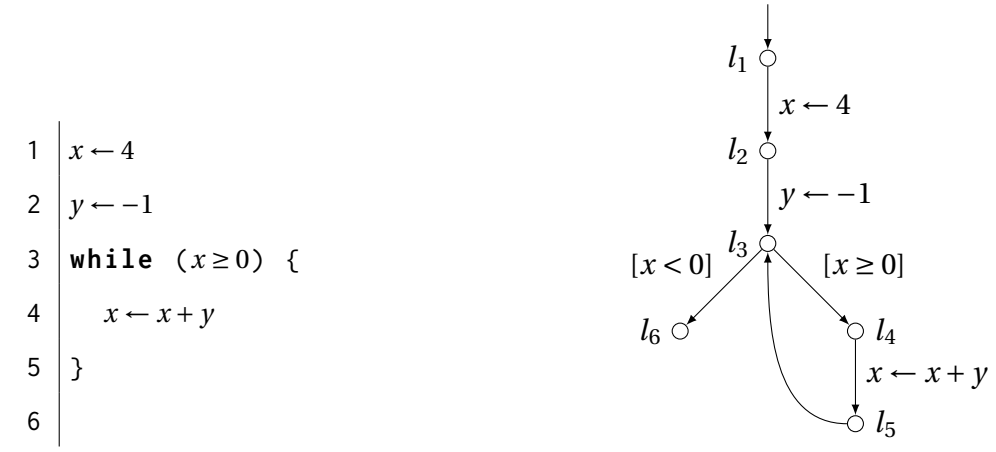
In this section, we introduce the formal notion of a program and its behaviour (an execution). We immediately note that our programs are mathematical models, and we will not define a formal correspondence between our theoretical notion of a program and the more practical notions, e.g., of an executable file. We assume though, that some correspondence exists and that our analysis can in the end be applied to practical programs. From the point of view of abstraction, we will assume that our programs are concrete, and are *not* produced by abstracting the statements of some original program. Thus, all abstraction will happen within the framework of abstract interpretation.

Programs

A *program* $\mathbb{P}$ is a graph $(\mathbb{L}, \mathfrak{l}_\vdash, \mathbb{E}, \mathfrak{c})$, where

- $\mathbb{L}$ is a *finite* set of vertices that are called *program locations*;
- $\mathfrak{l}_\vdash \in \mathbb{L}$ is a distinguished *initial location* where execution of the program starts;
- $\mathbb{E} \subseteq \mathbb{L} \times \mathbb{L}$ is a set of *edges* that define how control can flow in the program[1]; and

[1] We do allow edges to go into the initial location.



(a) Informal text in pseudocode.

(b) Formal program.

Figure 2.2: Informal text in pseudocode and a corresponding formal program.

- $\mathfrak{c} : \mathbb{E} \to \mathbb{A}$ labels edges with atomic statements, i.e., it specifies which statement gets executed when control moves from one location to another. We discuss atomic statements in more detain in Section 2.2.1.

A location without outgoing edges is called a *final location*. Intuitively, execution of the program terminates iff it reaches one of the final locations, and the computation cannot continue from that point. For a location $l \in \mathbb{L}$, the *successors* of l is the set

$$\mathrm{succ}(l) = \{l' \in \mathbb{L} \mid (l, l') \in \mathbb{E}\}.$$

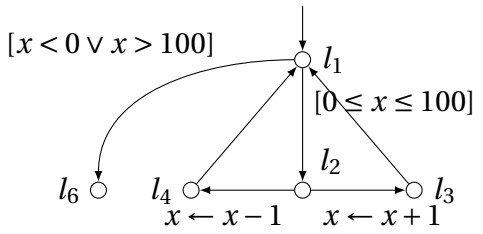
Note that by definition, for a pair of locations $l, l' \in \mathbb{L}$, at most one edge can go from l to l'. This simplifies the presentation, but does not restrict the class of programs that we can represent and analyse, as we can always introduce auxiliary locations and edges as a workaround (see Example 2.2 below). We also say that $\mathbb{P} = (\mathbb{L}, \mathfrak{l}_\vdash, \mathbb{E}, \mathfrak{c})$ is an *unstructured* program (in contrast to a notion of structured program introduced later in Section 2.5), as there are otherwise no restrictions on the form of the graph.

Example 2.1. Figure 2.2 gives an example of informal program text in C-like pseudocode and a corresponding formal program. Subscripts of program locations correspond to line numbers in program text. The initial location is $\mathfrak{l}_\vdash = l_1$, as indicated by an incoming pseudo-edge.

Example 2.2. Fig. 2.3 shows a loop with a non-deterministic branching in it. The meaning of the statement `if(*)` is that the program choses non-deterministically to

```
1 while (0 ≤ x ≤ 100) {
2   if (*)
3     x ← x + 1;
4   else
5     x ← x − 1;
6 }
```

(a) Informal text in C-like pseudocode.



(b) Possible formal program.

Figure 2.3: A loop with non-deterministic branching.

execute either the if-branch or the else-branch. After control reaches line 2 of the pseudocode (location l_2 of the formal program), it will be non-deterministically transferred either to line 3 (location l_3) or to line 5 (location l_4). Note how the increment/decrement edges cannot go directly from l_2 to l_1 because of our definition of the set of edges. Instead, we introduce auxiliary locations l_3 and l_4 and `skip`-edges from them to l_1. Another possible way to construct the program graph would be to label the edges (l_2, l_3) and (l_2, l_4) with `skip` and the edges (l_3, l_1) and (l_4, l_1) with increment and decrement.

Informally, we can say that our notion of a program corresponds to imperative programs without recursion where all non-recursive procedures were inlined. Academic researches often focus their attention on such programs, when the goal (as in our case) is to develop and study a novel analysis technique, rather than to apply an established approach to programs in a certain programming language.

States

At a given time during a run, the configuration of the program is described by the current program location and the current *memory state*. We denote the set of memory states by $\mathbb{M}$. We assume that there is a distinguished *error memory state* $\varepsilon \in \mathbb{M}$, but otherwise the structure of $\mathbb{M}$ is not restricted. It is common though that a program manipulates a finite set of variables which we denote by $\mathbb{V}$, and the memory state of the program is defined by their values. In this case, $\mathbb{M}$ maps every variable to its value. For example, the program in Fig. 2.2 uses a pair of variables $\mathbb{V} = \{x, y\}$. Assuming that both x and y take integer values, we can take for this program $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$. The error memory state represents a situation where one or more variables cannot be mapped

to a valid integer number. This can occur, e.g., as a result of executing the statement $x \leftarrow y/0$ which attempts to assign a new value to x, but results in division by zero.

The set $\mathbb{S} = \mathbb{L} \times \mathbb{M}$ is the set of *program states*. Thus, at any given time in a run, the configuration of the program can be described by an element of $\mathbb{S}$. We say that a program state $s \in \mathbb{S}$ is final iff $s = (l, m)$ for a final location $l \in \mathbb{L}$ and some memory state $m \in \mathbb{M}$. We say that a state (l, ε) (for some $l \in \mathbb{L}$) is an *error program state*.

Formulas

To make queries about the memory states, we assume that there is a language of *memory state formulas* Θ. The language of formulas is specific to the set of memory states $\mathbb{M}$ (but not uniquely defined by it) and thus we do not specify its structure, as we do not specify the structure of $\mathbb{M}$. We assume though that Θ is closed under negation, i.e. for every $\theta \in \Theta$, $\neg\theta \in \Theta$. For example, if $\mathbb{M}$ maps program variables to integer values: $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$ – then Θ can be the set of linear constraints over variables from $\mathbb{V}$, or the set of their Boolean combinations, etc.

We see a memory state formula θ as denoting a set of *non-error* memory states $[\![\theta]\!] \subseteq \mathbb{M} \setminus \{\varepsilon\}$. We say that a memory state $m \in \mathbb{M}$ satisfies θ if $m \in [\![\theta]\!]$, and the error memory state ε satisfies no formulas. For a memory-state formula θ and a set of memory states $M \subseteq \mathbb{M}$, the *value* of θ over M is a Kleene value defined as:

$$\mathrm{eval}(\theta, M) = \begin{cases} 1\,, \text{ if } M \subseteq [\![\theta]\!] \\ 0\,, \text{ if } M \neq \varnothing \wedge M \cap [\![\theta]\!] = \varnothing \\ {}^1\!/_2\,, \text{ otherwise} \end{cases}$$

That is, a formula evaluates to 1 in a *set* of memory states if all the memory states in the set satisfy the formula (including the case where the set is empty); to 0 if no memory states in the (non-empty) set satisfy the formula; and to ${}^1\!/_2$ if some memory states satisfy the formula and some do not.

2.2.1 Statements and Relations

The basic building block of a program is an *atomic statement*. This is an instruction that gets executed when control moves along an edge between a pair of locations. We

denote the set of *atomic statements* by $\mathbb{A}$. The formal meaning of an atomic statement[2] $C \in \mathbb{A}$ is defined by its *input-output relation* $T_{\mathbb{M}}(C) \subseteq \mathbb{M} \times \mathbb{M}$. That is, the effect of an atomic statement is to transform the memory state of a program. For a pair of memory states, $(m, m') \in T_{\mathbb{M}}(C)$, iff it is possible to produce m' by executing C from m. We assume that $\mathbb{A}$ includes the following statements.

(i) A passive statement `skip` that does not change the memory state:

$$T_{\mathbb{M}}(\texttt{skip}) = \{(m, m) \mid m \in \mathbb{M}\} = \Delta_{\mathbb{M}}$$

If an edge in a program is labeled by `skip`, we will not show the label in figures (e.g., the edge (l_5, l_3) in Fig. 2.2b).

(ii) An assumption statement $[\theta]$ for every memory-state formula θ. Assumption statements are used to represent branch and loop conditions. The statement does not change the memory state[3], but only allows to follow the edge from those memory states that satisfy the assumption formula (or from an error state):

$$T_{\mathbb{M}}([\theta]) = \{(m, m) \mid m \in [\![\theta]\!]\} \cup \{(\varepsilon, \varepsilon)\}$$

(iii) An assertion statement $\texttt{assert}(\theta)$ for every memory-state formula θ. An assertion makes the program fail when the conditions given by θ are not met. More formally, for a memory state formula θ,

$$T_{\mathbb{M}}(\texttt{assert}(\theta)) = \{(m, m) \mid m \in [\![\theta]\!]\} \cup \{(m, \varepsilon) \mid m \in \mathbb{M} \wedge m \notin [\![\theta]\!]\}$$

(iv) A set of domain-specific statements. Our example programs (like the one in Fig. 2.2) will usually be numeric and will manipulate a finite set of integer variables. Such programs will include assignment statements $x \leftarrow e$ and a non-deterministic assignment statement $x \leftarrow *$, where x is a variable and e is an expression. Intuitively, an assignment $x \leftarrow e$ sets the value of x to the value of x. A non-deterministic as-

[2] We use letter C, for *command*, to range over statements.

[3] This does not prevent us from modelling branching statements with side-effecting guards. For example, to model a conditional statement "`if` $(++x) \cdots$", we will introduce an edge labelled by the update statement $x \leftarrow x + 1$, followed by a branching point with two outgoing edges, labelled with assumptions $[x = 0]$ and $[x \neq 0]$.

signment $x \leftarrow *$ sets the value of x to a non-deterministically chosen value (possibly, a different one every time the statement is executed).

We assume that for the domain-specific atomic statements, their input-output relations are given and require that for every atomic statement $C \in \mathbb{A}$:

(i) If C is not an assumption statement then its input-output relation is left-total, i.e., for every memory state $m \in \mathbb{M}$, there exists at least one successor $m' \in \mathbb{M}$, s.t. $(m, m') \in T_{\mathbb{M}}(C)$.

(ii) Statements leave the error memory state unchanged: $(\varepsilon, \varepsilon) \in T_{\mathbb{M}}(C)$.

(iii) A program cannot recover from the error memory state: if $(\varepsilon, m) \in T_{\mathbb{M}}(C)$, then $m = \varepsilon$.

For example, let $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$ and the statement $C = x \leftarrow x + 1$. Then, $T_{\mathbb{M}}(C)$ can be defined as:

$$\{(m, \lambda v.\text{if } v = x \text{ then } m(v) + 1 \text{ else } f(v)) \mid m \in \mathbb{V} \to \mathbb{Z}\} \cup \{(\varepsilon, \varepsilon)\}$$

We note that atomic statements can be non-deterministic. That is, for an atomic statement C and a memory state $m \in \mathbb{M}$, there might exist two distinct memory states $m' \neq m''$, s.t. $(m, m') \in T_{\mathbb{M}}(C)$ and $(m, m'') \in T_{\mathbb{M}}(C)$.

An example of such statement is a non-deterministic assignment. Let $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$ and the statement $C = x \leftarrow *$. Then, $T_{\mathbb{M}}(C)$ can be defined as:

$$\{(m, \lambda v.\text{if } v = x \text{ then } n \text{ else } f(v)) \mid m \in \mathbb{V} \to \mathbb{Z} \text{ and } n \in \mathbb{Z}\} \cup \{(\varepsilon, \varepsilon)\}$$

Transition Relation on States

If we know the input-output relations of the statements of the program, we can define the *transition relation of the program on states* $T_{\mathbb{S}}(\mathbb{P}) \subseteq \mathbb{S} \times \mathbb{S}$:

$$\begin{aligned} T_{\mathbb{S}}(\mathbb{P}) = \{\big((l, m), (l', m')\big) \in \mathbb{S} \times \mathbb{S} \mid \big((l, l') \in \mathbb{E} \wedge (m, m') \in T_{\mathbb{M}}(\mathfrak{c}(l, l'))\big)\} \\ \cup \{\big((l, m), (l, m) \in \mathbb{S} \times \mathbb{S}\big) \mid l \text{ is final}\} \end{aligned}$$

That is, the transition relation consists of pairs of program states (s, s'), s.t. it is possible to reach s' either by executing a single program instruction (edge) from s, or by staying

in the same final state.

Example 2.3. Let us build the transition relation in states for the program $\mathbb{P}$ in Fig. 2.2, for the case when $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$. Let $m, m' \in \mathbb{V} \to \mathbb{Z}$. Then,

$$
\begin{aligned}
T_{\mathbb{S}}(\mathbb{P}) = \{&\big((l_1, m), (l_2, m')\big) \mid m'(x) = 4 \wedge m'(y) = m(y)\} \\
&\cup \{\big((l_1, \varepsilon), (l_1, \varepsilon)\big)\} \\
&\cup \{\big((l_2, m), (l_3, m')\big) \mid m'(y) = -1 \wedge m'(x) = m(x)\} \\
&\cup \{\big((l_2, \varepsilon), (l_2, \varepsilon)\big)\} \\
&\cup \{\big((l_3, m), (l_4, m')\big) \mid m(x) \geq 0 \wedge m' = m\} \\
&\cup \{\big((l_3, m), (l_6, m')\big) \mid m(x) < 0 \wedge m' = m\} \\
&\cup \{\big((l_3, \varepsilon), (l_3, \varepsilon)\big)\} \\
&\cup \{\big((l_4, m), (l_5, m')\big) \mid m'(x) = m(x) + m(y) \wedge m'(y) = m(y)\} \\
&\cup \{\big((l_4, \varepsilon), (l_4, \varepsilon)\big)\} \\
&\cup \{\big((l_5, m), (l_3, m')\big) \mid m' = m\} \\
&\cup \{\big((l_5, \varepsilon), (l_5, \varepsilon)\big)\} \\
&\cup \{\big((l_6, m), (l_6, m')\big) \mid m' = m\} \\
&\cup \{\big((l_6, \varepsilon), (l_6, \varepsilon)\big)\}
\end{aligned}
$$

One can spot a pattern here. The transition relation is made of a number of disjuncts, each corresponding to a single edge in the program.

2.2.2 Executions

We can now introduce the formal notion of the behaviour of a program. Intuitively, as the program runs, it goes through a sequence of configurations. Thus, a single run of a program will be described by a sequence of program states[4]. For that, we use an *infinite* sequence of states, regardless of whether or not the run terminates in a final state. This allows to uniformly describe terminating and non-terminating runs.

[4]There might exist different possible program runs, due to non-determinism. First, by definition, all programs start in one of the many possible initial states (the set of initial states is $\{l_\vdash\} \times \mathbb{M}$). Additionally, some program statements may be non-deterministic (e.g., those that generate arbitrary numbers, receive user input, etc).

Path A *path* is a pair $(p, i) \in \mathbb{L}^{\mathbb{N}} \times \mathbb{N}$, where $p = \langle l_0, l_1, l_2, \ldots \rangle \in \mathbb{L}^{\mathbb{N}}$ is an *infinite* sequence of locations, and $i \geq 0$ is the (current) *position*. Intuitively, a path consists of the current program location, the locations that were visited in the past, and the locations that will be visited in the future. We denote the set of all paths by $\Pi = \mathbb{L}^{\mathbb{N}} \times \mathbb{N}$. For a path $\pi = (p, i) \in \Pi$, $p_{(0)}$ and $\pi_{(0)}$ denote the first location in the path; $p_{(j)}$ and $\pi_{(j)}$ – the j+1-th location.

There are no constraints on which locations appear in a path and in which order. For example, for the program in Fig. 2.3, all the following objects are paths:

$$(\langle l_1, l_2, l_4, l_1, l_6^{\mathbb{N}} \rangle, 0)$$
$$(\langle (l_1, l_2, l_3, l_1, l_2, l_4)^{\mathbb{N}} \rangle, 100)$$
$$(\langle l_6, (l_1, l_3, l_2)^{\mathbb{N}} \rangle, 81)$$

Trace A *trace* is a pair $(t, i) \in \mathbb{S}^{\mathbb{N}} \times \mathbb{N}$, where $t = \langle s_0, s_1, s_2, \ldots \rangle \in \mathbb{S}^{\mathbb{N}}$ is an infinite sequence of program states, and $i \geq 0$ is the (current) position. Intuitively, a trace consists of the current program state, the states that were visited in the past, and the states that will be visited in the future. We denote the set of traces by Σ. For a trace $\tau = (t, i) \in \Sigma$, $t_{(0)}$ and $\tau_{(0)}$ denote the first state of the trace; $t_{(j)}$ and $\tau_{(j)}$ denote the j+1-th state.

We can also lift indexing to finite sequences of states. For a finite sequence of program states $\tau' = \langle (l_0, m_0), \cdots, (l_k, m_k) \rangle \in \mathbb{S}^*$, $\tau'_{(i)} = (l_i, m_i)$ for $i = 0..k$ and $\tau'_{(-1)} = (l_k, m_k)$.

For a trace $\tau \in \Sigma$, its path $\mathfrak{p}(\tau) \in \Pi$ is produced by removing information about the memory states:

$$\text{For } \tau = \big(\langle (l_0, m_0), (l_1, m_1), (l_2, m_2), \ldots \rangle, i\big) \in \Sigma,$$
$$\mathfrak{p}(\tau) = \big(\langle l_0, l_1, l_2, \ldots \rangle, i\big) \in \Pi$$

We say that a trace is *terminating* iff there exists $j \geq 0$, a final location $l \in \mathbb{L}$, and a memory state $m \in \mathbb{M}$, s.t. for every $k \geq j$, $\tau_{(k)} = (l, m)$. We say that a trace is non-terminating iff it is not terminating.

Again, there are no constraints on which locations and memory states appear in a path and in which order. For example, for the program in Fig. 2.3, the following object

is a terminating trace:

$$(\langle (l_1, x \mapsto 0), (l_2, x \mapsto 0), (l_4, x \mapsto -1), (l_1, x \mapsto -1), (l_6, x \mapsto -1)^{\mathbb{N}} \rangle, 0)$$

and the following objects are non-terminating traces:

$$(\langle \big((l_1, x \mapsto 0), (l_2, x \mapsto 0), (l_3, x \mapsto 1), (l_1, x \mapsto 1), (l_2, x \mapsto 1), (l_4, x \mapsto 0)\big)^{\mathbb{N}} \rangle, 100)$$
$$(\langle (l_6, \varepsilon), \big((l_1, \varepsilon), (l_3, \varepsilon), (l_2, \varepsilon)\big)^{\mathbb{N}} \rangle, 81)$$

Execution Given a program $\mathbb{P}$, not every trace can be produced by it. We distinguish the following kinds of traces:

(i) A trace $(t, i) \in \Sigma$ is an *execution prefix* iff $t_{(0)} = (l_\vdash, m)$ for some memory state $m \in \mathbb{M}$, and for every j, s.t. $0 \leq j < i$, $(t_{(j)}, t_{(j+1)}) \in T_{\mathbb{S}}(\mathbb{P})$. Intuitively, for an execution prefix (t, i), the prefix of t up to position i is produced by starting in the initial location in some memory state and making i steps through the program.

(ii) A trace $(t, i) \in \Sigma$ is an *execution postfix* iff for every $j \geq i$, $(t_{(j)}, t_{(j+1)}) \in T_{\mathbb{S}}(\mathbb{P})$.

(iii) A trace $\tau \in \Sigma$ is a *semi-execution* of $\mathbb{P}$ iff for every $j \geq 0$, $(\tau_{(j)}, \tau_{(j+1)}) \in T_{\mathbb{S}}(\mathbb{P})$.

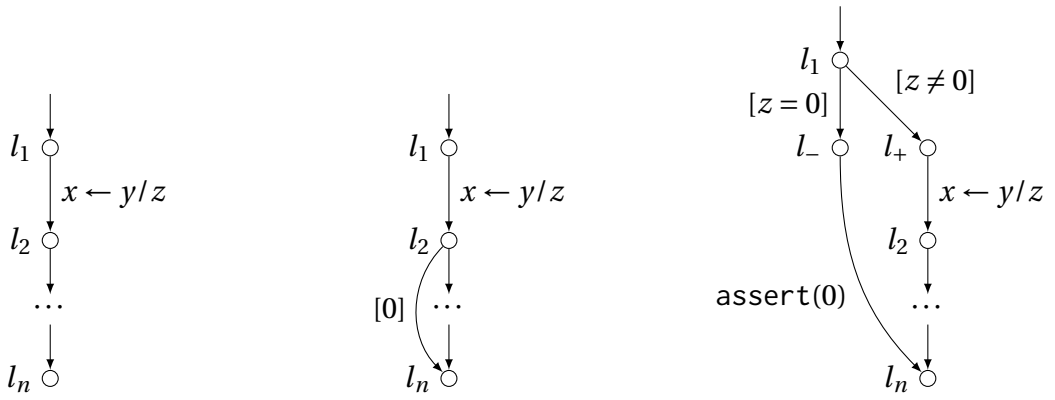
(iv) Finally, a trace $\tau \in \Sigma$ is an *execution*, if it is a semi-execution and $\tau_{(0)} = (l_\vdash, m)$ for some memory state $m \in \mathbb{M}$.

Intuitively, an execution has as its first component a sequence of program states that is produced by starting in the initial program location in some memory state, and running the program either infinitely, producing a non-terminating execution[5], or until it terminates in a final location, producing a terminating one. For a terminating execution postfix (hence, also for a semi-execution and for an execution) τ, we denote the final state of the execution postfix by $\tau_{(\dashv)}$, i.e., for a terminating execution $\tau = \langle (l_0, m_0), \cdots, (l_k, m_k)^{\mathbb{N}} \rangle$, $\tau_{(\dashv)} = (l_k, m_k)$.

For example, for the program in Fig. 2.3, the following object is a terminating execution with the final state $(l_6, x \mapsto 1)$:

$$(\langle (l_1, x \mapsto 0), (l_2, x \mapsto 0), (l_4, x \mapsto -1), (l_1, x \mapsto -1), (l_6, x \mapsto -1)^{\mathbb{N}} \rangle, 0)$$

[5]Note that the only way to produce a non-terminating execution is to infinitely execute some set of edges. Informally, this means that we assume a single atomic statement to always terminate.



(a) No explicit error handling.

(b) The program may abort from an error state.

(c) The program must abort from an error state.

Figure 2.4: Ways to handle errors in an unstructured program.

and the following object is a non-terminating execution:

$$(\langle\big((l_1, x \mapsto 0), (l_2, x \mapsto 0), (l_3, x \mapsto 1), (l_1, x \mapsto 1), (l_2, x \mapsto 1), (l_4, x \mapsto 0)\big)^{\mathbb{N}}\rangle, 100)$$

Transition Relation on Traces

Then, we lift the program transition relation to paths and traces. The transition relation on paths is

$$T_\Pi(\mathbb{P}) = \{\big((p,i),(p,i+1)\big) \in \Pi \times \Pi \mid (p_{(i)}, p_{(i+1)}) \in \mathbb{E}\}$$

The transition relation on traces is

$$T_\Sigma(\mathbb{P}) = \{\big((t,i),(t,i+1)\big) \in \Sigma \times \Sigma \mid (t_{(i)}, t_{(i+1)}) \in T_{\mathbb{S}}(\mathbb{P})\}$$

Note how the transition relations preserve the sequence of locations (p in the definition) and of program states (t). That is, in a path or in a trace, both the past and the future are pre-determined and do not change when taking a step along the transition relation.

For a program $\mathbb{P}$, the meaning (or *semantics*) of the program $[\![\mathbb{P}]\!] \subseteq \Sigma$ is the set of all executions of $\mathbb{P}$.

Errors in Executions

The way atomic statements transform the error memory state ensures that once an execution reaches an error state ($(l, \varepsilon) \in \mathbb{S}$ for some $l \in \mathbb{L}$), from that point it only visits error states and follows some (arbitrary) valid path. This corresponds to how errors manifest in low-level languages: executing an erroneous statement (e.g., dereferencing an invalid pointer) may corrupt the memory state in an unexpected way. Otherwise, the way an unstructured program reacts to errors is not restricted and is defined by the program itself.

For example, if we want to model the situation where a program *may* abort (i.e., terminate pre-maturely) when it transitions to an error state, we can do the following. For every location $l_i \in \mathbb{L}$ from which the program may abort, we add an edge that leads to some final location and is labeled by the statement [0] (assume false). This ensures that if the execution reaches l_i in the error memory state, it may immediately transition to the final location (but must continue normally otherwise, as $T_{\mathbb{M}}([0]) = \{(\varepsilon, \varepsilon)\}$). This is shown in Fig. 2.4b.

Sometimes we want to model the situation where atomic statements fail deterministically, and failure *must* cause the program to abort. This corresponds to the behaviour of some managed languages (e.g., Java) where some actions, e.g., using an invalid reference, necessarily result in an exception. We can implement this by explicitly checking the pre-condition of every statement that may fail (has ε as the successor of some non-error memory state). This is shown in Fig. 2.4c.

The way structured programs (they will be introduced later in Section 2.5) will react to errors is more restricted.

That said, most part of this work (the remaining part of this Chapter and Chapters 3–4) focuses on error-free non-terminating executions[6] and on the notion of recurrent set, which by definition does not include error states. Errors are revisited and get more attention in Chapter 6.

[6] This does not mean that we will only consider error-free programs. This means that we will be looking for a non-terminating execution that does not visit the error memory state.

2.3 Introduction to Abstract Interpretation

In this section, first, we introduce *abstract interpretation* – a systematic approach to computing approximate properties of mathematical objects, where these properties are expressed as fixed points of monotone functions. The theory of abstract interpretation was initially developed by Patrick and Radhia Cousot in the context of static program analysis [CC77], but it is applicable in other contexts as well (formal languages, graph algorithms, etc [Cou15]). Later in the section, we demonstrate how we can use abstract interpretation to reason about the set of executions of a program.

Concrete Domain

Given some mathematical object (e.g., a program), we assume that properties of the object come from a partially ordered set $\mathscr{L}_\flat$ which is called *concrete domain*. The term *domain* is usually ambiguous and means a kind of partially ordered set that is required in a given context (w.r.t. existence of specific upper or lower bounds, greatest or least element, etc). Often (an in this work as well) $\mathscr{L}_\flat$ is assumed to be a complete lattice $\mathscr{L}_\flat\langle\sqsubseteq_\flat, \sqcup_\flat, \sqcap_\flat, \bot_\flat, \top_\flat\rangle$. Then, we characterize the property of interest as the least or greatest fixed point of some monotone function on $\mathscr{L}_\flat$.

Example 2.4. For example, the program in Fig. 2.5 has a single variable x, which we assume to take integer values. The program initializes x with 0 and then increments it in a loop zero or more times[7]. We may want to know what can be the value of x when the loop terminates. The answer will not be a single value, but rather a subset of $\mathbb{Z}$ of possible values. That is, we may take $\mathscr{L}_\flat = \mathscr{P}(\mathbb{Z})\langle\subseteq, \cup, \cap, \varnothing, \mathbb{Z}\rangle$ to be the concrete domain (since the program is error-free, we do not have to include the error state in the domain). We may notice that the set in question is the smallest set that includes 0 (for the case when the loop makes 0 iterations and terminates) and $n+1$ for every n in the set (for the case when the loop first makes n iterations and terminates after the $n+1$-th one). This can be expressed as the least fixed point:

$$\mathrm{lfp}_{\subseteq}\lambda N.(\{0\} \cup \{n+1 \mid n \in N\}) = \{n \in \mathbb{Z} \mid n \geq 0\} \tag{2.1}$$

[7] Here, we write `while(*)` to denote a loop, s.t. the program non-deterministically decides whether to exit it or to make another iteration.

```
1 | x ← 0
2 | while (*) {
3 |     x ← x + 1
4 | }
```

Figure 2.5: A loop that increments x an unknown number of times.

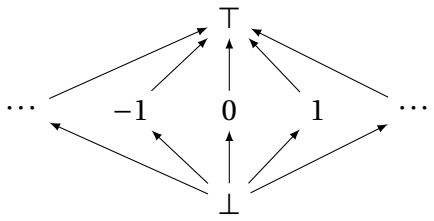


Figure 2.6: Constant propagation abstract domain.

In the context of analyzing programs (in the sense of Section 2.2), $\mathscr{L}_\flat$ is usually the powerset $\mathscr{P}(\Sigma)$ of the set of program traces. That is, the goal of program analysis is usually to approximately compute for a program $\mathbb{P}$ a specific set of traces: e.g., the set of all program executions $[\![\mathbb{P}]\!]$, the set of error-free executions, the set of non-terminating executions, etc.

Abstract Domain

Usually, a fixed point of a function in the concrete domain, as in (2.1), cannot be computed directly (e.g., via iterating the function on the least or greatest element [Kle87]), and its value cannot be directly represented by a computer. The approach of abstract interpretation is to compute a *sound approximation* of the concrete fixed point in the following way. We introduce the *abstract domain* $\mathbb{D}_\sharp$, where an abstract element $d \in \mathbb{D}_\sharp$ represents some concrete element $\gamma(d) \in \mathscr{L}_\flat$. The function $\gamma : \mathbb{D}_\sharp \to \mathscr{L}_\flat$ that maps abstract elements to concrete elements they represent is called *concretization function* and is required to be monotone: for $d_1, d_2 \in \mathbb{D}_\sharp$,

$$d_1 \sqsubseteq_\sharp d_2 \Rightarrow \gamma(d_1) \sqsubseteq_\flat \gamma(d_2)$$

Usually[8] $\mathbb{D}_\sharp$ consists of finite representations of a subset of $\mathscr{L}_\flat$. For example, in an *interval domain*, an element is a range of numerical values, or in a more general setting – a map from program variables to their ranges. An interval finitely represents a potentially infinite set of numbers, but not every set of numbers can be represented by an interval. Another example is a *polyhedral domain* where an element is a conjunction of linear inequalities over program variables.

[8] Not necessarily, though. In a theoretical discussion, both $\mathscr{L}_\flat$ and $\mathbb{D}_\sharp$ may have elements that do not have finite representations. This will be the case later when we will discuss set-of-states abstraction.

For simplicity of presentation, we will assume $\mathbb{D}_\sharp$ in this section to be a complete lattice[9] $\mathbb{D}_\sharp\langle\sqsubseteq_\sharp, \sqcup_\sharp, \sqcap_\sharp, \bot_\sharp, \top_\sharp\rangle$. In general, in literature, by *domain*, authors (and we as well) mean a partial order, where, depending on the analysis that is performed, there exist: (i) specific elements or operations, like the least and greatest elements, upper and lower bounds, etc (In Chapters 3–6, we will always specify what operations the domain should support); and (ii) limits of infinite ascending or descending chains.

For an abstract element $d \in \mathbb{D}_\sharp$ and a concrete element $l \in \mathscr{L}_\flat$, we say that d *over-approximates* l if $\gamma(d) \sqsupseteq_\flat l$, and that d *under-approximates* l if $\gamma(d) \sqsubseteq_\flat l$. The notion of soundness depends on the problem and property at hand. For example, we may wish to know whether all executions of a program are error free. Then, we will compute an over-approximation of the set of program executions. If all the executions in this set are error-free, then all actual program executions are. We will always indicate which approximation we consider sound for a given problem.

Abstract interpretation provides a systematic way to over-approximate properties characterized as least fixed points and under-approximate properties characterized as greatest fixed points. Given a function $F_\flat : \mathscr{L}_\flat \to \mathscr{L}_\flat$ on the concrete domain, we say that a function $F_\sharp : \mathbb{D}_\sharp \to \mathbb{D}_\sharp$ on the abstract domain *over-approximates* $F_\flat$ if for every $d \in \mathbb{D}_\sharp$, $\gamma(F_\sharp(d)) \sqsupseteq_\flat F_\flat(\gamma(d))$. Similarly, $F_\sharp$ *under-approximates* $F_\flat$ if for every $d \in \mathbb{D}_\sharp$, $\gamma(F_\sharp(d)) \sqsubseteq_\flat F_\flat(\gamma(d))$. It is not necessary (for the purpose of over- or under-approximation) for $F_\sharp$ to be monotone. In program analysis, transfer functions are produced from program itself and thus $F_\flat$ represents the concrete program in the characterization of the property, while we can think of $F_\sharp$ as representing an abstracted program. This way, abstraction of program statements is performed.

Over-Approximating an LFP

Let us assume that $F_\sharp$ over-approximates $F_\flat$. Let us also assume that we find an element $d_{\text{lim}} \in \mathbb{D}_\sharp$, s.t. $F_\sharp(d_{\text{lim}}) \sqsubseteq_\sharp d_{\text{lim}}$. It follows that $F_\flat(\gamma(d_{\text{lim}})) \sqsubseteq_\flat \gamma(F_\sharp(d_{\text{lim}})) \sqsubseteq_\flat \gamma(d_{\text{lim}})$. That is (from Knaster-Tarski theorem[10]), $\gamma(d_{\text{lim}}) \sqsupseteq_\flat \text{lfp}_{\sqsubseteq_\flat} F_\flat$, and d_{lim} over-approximates the

[9] Not all useful abstract domains are complete lattices. A more rigorous approach would be to assume the domain to be a join-semilattice when it is used for over-approximation and a meet-semilattice when it is used for under-approximation. We believe though that this does not benefit the presentation in this chapter, and the extra details will only be distracting.

[10] Informally, it states that the least fixed point of a function is the infimum of the set, where the function is contracting.

least fixed point of $F_\flat$.

To find d_{lim}, we build an ascending chain of elements[11]:

$$d_0 \sqsubseteq_\sharp d_1 \sqsubseteq_\sharp d_2 \sqsubseteq_\sharp \cdots, \text{ where}$$
$$d_0 = \bot_\sharp$$
$$d_i = d_{i-1} \sqcup_\sharp F_\sharp(d_{i-1}), \text{ for } i \geq 1$$

A number of domains satisfy the *ascending chain condition* (we also say that such domains have *finite height*), and no infinite ascending chain can be strictly ascending. That is, for every infinite ascending chain $d_0 \sqsubseteq_\sharp d_1 \sqsubseteq_\sharp d_2 \cdots$ there exists $k \geq 0$, s.t. for $i \geq k$, $d_{i+1} = d_i$ (which also implies that $F_\sharp(d_i) \sqsubseteq_\sharp d_i$). For such domains, in finite number of k steps we find $d_{\text{lim}} = d_k$ and say that it is the *stable limit* of the chain.

In static analysis, a number of finite height domains is used. For example, one performs constant propagation, i.e. when one wishes to identify the expressions in the program, s.t. their value is constant and can be computed at compile time, one uses the domain $\mathbb{D}_\sharp = \mathbb{Z} \cup \{\bot, \top\}$ (assuming for simplicity that the expressions take integer values) ordered by $\sqsubseteq$, s.t. $d \sqsupseteq \bot$, $d \sqsubseteq \top$ for every $d \in \mathbb{D}_\sharp$, and distinct numbers are incomparable (see Fig. 2.6). Top ($\top$) means that the value of an expression cannot be computed at compile time (i.e., absence of definite answer), and bottom ($\bot$) means that an expression is not reachable in the program. Another example of a domain of finite height is the domain of bounded 3-valued structures [SRW02]. The domain uses sets of Kleene logic models of certain form (s.t. there is a finite number of them) to represent the contents of the dynamic memory of a program. Some classes of finite-height domains are well studied and admit efficient implementation of analyses [RHS95; SRH96].

That said, many numeric analyses use domains of infinite height. For example, interval domain is usually implemented in a way that admits infinite ascending and

[11] The fact for a concrete transfer function $F_\flat$, there exists an over-approximate abstract function $F_\sharp$, contains an implicit assumption on the abstract domain $\mathbb{D}_\sharp$. Namely, it requires that for every element of a concrete chain (that we can build out of elements of $\mathscr{L}_\flat$ using $F_\flat$), there exist an over-approximating element of $\mathbb{D}_\sharp$. Practical abstract domain usually have a stronger property: for every concrete element $l \in \mathscr{L}_\flat$, there will exist a pair of abstract elements $d_o, d_u \in \mathbb{D}_\sharp$ that over- and under-approximate l: $\gamma_\sharp(d_u) \subseteq l \subseteq \gamma_\sharp(d_o)$.

descending chains, e.g.:

$$[0;1] \sqsubseteq [0;2] \sqsubseteq [0;3] \sqsubseteq \cdots$$

or

$$[0;+\infty) \sqsupseteq [1;+\infty) \sqsupseteq [2;+\infty) \sqsupseteq \cdots$$

For such domain we introduce a partial function $\triangledown_\sharp : \mathbb{D}_\sharp \rightharpoonup \mathbb{D}_\sharp$ that is called *widening* and has the following properties:

(i) For $d_1, d_2 \in \mathbb{D}_\sharp$, if $d_1 \triangledown_\sharp d_2$ is defined[12] then $d_1 \sqsubseteq_\sharp d_1 \triangledown_\sharp d_2$ and $d_2 \sqsubseteq_\sharp d_1 \triangledown_\sharp d_2$;

(ii) For every increasing chain $g_0 \sqsubseteq_\sharp g_1 \sqsubseteq_\sharp g_2 \sqsubseteq_\sharp \cdots$, if the following chain is defined:

$$\begin{aligned} &d_0 \sqsubseteq_\sharp d_1 \sqsubseteq_\sharp d_2 \sqsubseteq_\sharp \cdots, \text{ where} \\ &d_0 = d_0 \\ &d_i = d_{i-1} \triangledown_\sharp g_i, \text{ for } i \geq 1 \end{aligned}$$

then it is not strictly increasing.

Usually, in a numeric abstract domain $\mathbb{D}_\sharp$, widening $d_1 \triangledown_\sharp d_2$ is defined when $d_1 \sqsubseteq_\sharp d_2$. Thus, in a domain of infinite height, to find d_{lim}, we instead build an ascending chain of elements:

$$\begin{aligned} &d_0 \sqsubseteq_\sharp d_1 \sqsubseteq_\sharp d_2 \sqsubseteq_\sharp \cdots, \text{ where} \\ &d_0 = \bot_\sharp \\ &d_i = d_{i-1} \triangledown_\sharp \left(d_{i-1} \sqcup_\sharp F_\sharp(d_{i-1})\right), \text{ for } i \geq 1 \end{aligned}$$

From the properties of widening, it follows that the chain is not strictly increasing [Bag+05]. Therefore, there exists $k \geq 0$, s.t. for $i \geq k$, $d_{i+1} = d_i$ (which also implies that $F_\sharp(d_i) \sqsubseteq_\sharp d_i$), and in finite number of k steps we find $d_{\text{lim}} = d_k$.

Example 2.5. For example, let us compute an over-approximation of the fixed point (2.1) in the interval domain. To over-approximate the concrete transfer function $F_\flat =$

[12] We prefer to follow the definition of [Bag+05], which admits that widening operator may not be total. Usually for numeric domains, $d_1 \triangledown_\sharp d_2$ is defined when $d_1 \sqsubseteq_\sharp d_2$. In this case one could define an alternative total widening operator $d_1 \triangledown'_\sharp d_2 = d_1 \triangledown_\sharp (d_1 \sqcup_\sharp d_2)$.

$\lambda N.\{0\} \cup \{n+1 \mid n \in N\}$, we take

$$F_\sharp = \lambda d.[0;0] \sqcup (d+1)$$

where + operation increases the bounds of an interval by a given constant. First, let us build an ascending chain without widening:

$$\begin{aligned}
d_0 &= \bot \\
d_1 &= \bot \sqcup [0;0] \sqcup (\bot + 1) = [0;0] \\
d_2 &= [0;0] \sqcup [0;0] \sqcup ([0;0] + 1) \\
&= [0;0] \sqcup [0;0] \sqcup [1;1] = [0;1] \\
d_3 &= [0;1] \sqcup [0;0] \sqcup ([0;1] + 1) = [0;2] \\
d_4 &= [0;2] \sqcup [0;0] \sqcup ([0;2] + 1) = [0;3] \\
d_5 &= \cdots
\end{aligned}$$

This results in an infinite strictly increasing chain. For numeric domains, widening is typically defined in a way that $d_1 \triangledown d_2$ is formed of the bounds or constraints that are shared by d_1 and d_2 (i.e., stable constraints). In other words, widening removes unstable constraints. Now, let us build the ascending chain with widening:

$$\begin{aligned}
d_0 &= \bot \\
d_1 &= \bot \triangledown \big(\bot \sqcup [0;0] \sqcup ([0;0] + 1)\big) = [0;0] \\
d_2 &= [0;0] \triangledown [0;1] = [0;1] \\
d_3 &= [0;1] \triangledown [0;2]
\end{aligned}$$

This may be good time to remove the unstable bound

$$\begin{aligned}
&= [0;+\infty) \\
d_4 &= [0;+\infty) = d_{\text{lim}}
\end{aligned}$$

In this case, we were able to compute the exact interval representation $[0;+\infty)$ of the concrete least fixed point value $\{n \in \mathbb{Z} \mid n \geq 0\}$, but this will not be the case in general. First, the concrete least fixed point might not have exact representation as an abstract element. Second, widening will often produce result that is above (w.r.t. $\sqsubseteq_\sharp$)

the exact or best representation of the concrete least fixed point even when the exact or best representation exists.

Practical definitions of widening include heuristics that work around typical cases of over-approximating too much. A common technique is *widening delay*, when first few elements of the ascending chain are computed without widening (thus, to be formal, we may want to define widening as a partial function from *sequences* to single elements $\triangledown_\sharp : \mathbb{D}_\sharp^* \rightharpoonup \mathbb{D}_\sharp$). A number of heuristics specific to polyhedral domain are described in [Bag+05]. It is also sometimes possible to improve the over-approximation produced by widening, by computing a descending chain of over-approximants where convergence is due to *narrowing* operator. In this work, we regard this as an auxiliary technique which is not strictly required for soundness or computability, and we do not discuss it further.

Under-Approximating a GFP

The approach to under-approximating a greatest fixed point is dual. For a concrete monotone transfer function $F_\flat$ and its abstract under-approximation $F_\sharp$, if we find an element $d_{\text{lim}} \in \mathbb{D}_\sharp$, s.t. $F_\sharp(d_{\text{lim}}) \sqsubseteq_\sharp d_{\text{lim}}$, then $\gamma(d_{\text{lim}}) \sqsubseteq_\flat \text{lfp}_{\sqsubseteq_\flat} F_\flat$. We find d_{lim} as the stable limit of the descending chain:

$$
\begin{aligned}
&d_0 \sqsupseteq_\sharp d_1 \sqsupseteq_\sharp d_2 \sqsupseteq_\sharp \cdots, \text{ where} \\
&d_0 = \top_\sharp, \text{ and for } i \geq 1 \\
&d_i = d_{i-1} \sqcap_\sharp F_\sharp(d_{i-1}) \text{ when } \mathbb{D}_\sharp \text{ satisfies the descending chain condition} \\
&\text{or} \\
&d_i = d_{i-1} \mathbin{\underline{\triangledown}_\sharp} \left(d_{i-1} \sqcap_\sharp F_\sharp(d_{i-1})\right), \text{ otherwise}
\end{aligned}
$$

The operation $\underline{\triangledown}_\sharp$ is called *lower widening* or *dual widening*, and it ensures convergence of descending chains. For polyhedra, lower widening is usually based on removing unstable generators (points, rays, etc.) [Min13].

In this work, we will not be under-approximating greatest fixed points directly as shown above. In practice, many abstract domains are designed in the first place to support over-approximation, and it may be difficult to produce an under-approximate transfer function $F_\sharp$. We will be using lower widening though.

Galois Connection and Exact Fixed Point Abstraction

For a concrete domain $\mathscr{L}_\flat$, an abstract domain $\mathbb{D}_\sharp$, and a concretization function γ, it may be the case that every element of $\mathscr{L}_\flat$ has a unique best abstraction in $\mathbb{D}_\sharp$. We denote the best abstraction of $l \in \mathscr{L}_\flat$ by $\alpha(l) \in \mathbb{D}_\sharp$ where $\alpha : \mathscr{L}_\flat \to \mathbb{D}_\sharp$ is the *abstraction function*. When an abstraction function satisfies the property

$$\forall l \in \mathscr{L}_\flat, d \in \mathbb{D}_\sharp.\ \alpha(l) \sqsubseteq_\sharp d \Leftrightarrow l \sqsubseteq_\flat \gamma(d)$$

we say that α and γ form a *Galois connection* between $\mathscr{L}_\flat$ and $\mathbb{D}_\sharp$ and write

$$\mathscr{L}_\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D}_\sharp$$

We should note that some important abstract domains may not form a Galois connection with the corresponding concrete domain. For example, for a 2-dimensional rational sphere $\{(x, y) \in \mathbb{Q}^2 \mid x^2 + y^2 \leq r^2\}$ (for some $r \geq 0$), there exists the best but inexact representation in the interval domain: $\langle x : [-r; r], y : [-r; r]\rangle$; but there is no best representation in polyhedral domain (i.e., in form of a finite conjunction of linear inequalities). This shows that polyhedral domain does not form a Galois connection with a concrete domain that maps program variables to rational or real values.

Galois connections have a number of interesting properties, and we note the ability to compute under certain conditions the *exact fixed point abstraction*. If the concrete transfer function $F_\flat : \mathscr{L}_\flat \to \mathscr{L}_\flat$ and its abstract approximation $F_\sharp : \mathbb{D}_\sharp \to \mathbb{D}_\sharp$ are monotone and are such that $\alpha \circ F_\flat = F_\sharp \circ \alpha$, then

$$\alpha(\mathrm{lfp}_{\sqsubseteq_\flat} F_\flat) = \mathrm{lfp}_{\sqsubseteq_\sharp} F_\sharp$$

This is a standard result [CC79, theorem 7.1.0.4]. In other words, if we want to compute a summarized property of a mathematical object that can be represented as an element of the abstract domain $\mathbb{D}_\sharp$, we may perform this computation directly in $\mathbb{D}_\sharp$ (as opposed to computing in $\mathscr{L}_\flat$ and then abstracting) without loss of information. For example, if we are only interested in which states a program visits (and not interested, in which order), we may perform the computation over sets of program states, and not over sets of program executions.

For the greatest fixed points, this does not seem to hold in general, and we can only make a weaker statement.

Lemma 2.1. Let $\mathscr{L}_\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D}_\sharp$. Also, let the concrete transfer function $F_\flat : \mathscr{L}_\flat \to \mathscr{L}_\flat$ and its abstract approximation $F_\sharp : \mathbb{D}_\sharp \to \mathbb{D}_\sharp$ be monotone and such that $\alpha \circ F_\flat = F_\sharp \circ \alpha$. Then

$$\alpha(\mathrm{gfp}_{\sqsubseteq_\flat} F_\flat) \sqsubseteq_\sharp \mathrm{gfp}_{\sqsubseteq_\sharp} F_\sharp$$

Proof Idea. Via Knaster-Tarski theorem. We give the full proof in Appendix 2.A. □

Still, exact abstraction of a greatest fixed point *will* hold in a particular interesting case in Section 2.4, when we will move with non-termination analysis from the domain of traces to the domain of states.

2.3.1 Analyzing Programs in the Domain of Traces

In this work, our goal is to reason about the behaviour of programs, i.e., sets of their executions. For that, we use the domain of (sets of) traces $\mathscr{P}(\Sigma)\langle \subseteq, \cup, \cap, \varnothing, \Sigma \rangle$ as the concrete domain. More specifically, program properties will be sets of program *executions*, but their characterizations and intermediate computation steps will manipulate sets of traces.

The transfer functions that we use to characterize program properties will be post- and pre-conditions w.r.t. different transition relations.

Given some set S_0, a transition relation $T \subseteq S_0 \times S_0$, and a subset $S \subseteq S_0$, we define the *post-condition*, *predecessor*, and (weakest liberal) *pre-condition* transfer functions (or *transformers*) respectively as:

$$\begin{aligned} \mathrm{post}(T, S) &= \{s' \in S_0 \mid \exists s \in S.\ (s, s') \in T\} \\ \mathrm{pre}(T, S) &= \{s \in S_0 \mid \exists s' \in S.\ (s, s') \in T\} \\ \mathrm{wp}(T, S) &= \{s \in S_0 \mid \forall s' \in S_0.\ (s, s') \in T \Rightarrow s' \in S\} \end{aligned} \tag{2.2}$$

Intuitively, for a set S, if we pick an element of S and make a step along the transition relation T, we will obtain some element of $\mathrm{post}(T, S)$. If we make a step along T in backward direction, we will obtain some element of $\mathrm{pre}(T, S)$. Finally, if we pick an

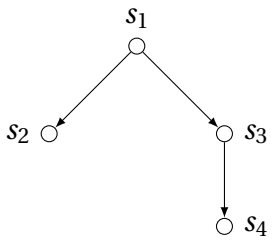


Figure 2.7: Example of a transition relation.

element of $\mathrm{wp}(T, S)$ and (if possible) make a step along the transition T, we will obtain some element of S. Let us now look at a small example.

Example 2.6. Let $S_0 = \{s_1, \cdots, s_4\}$, $T = \{(s_1, s_2), (s_1, s_3), (s_3, s_4)\}$, and $S = \{s_2, s_3\}$. This is shown in Fig. 2.7. Then

$$\mathrm{post}(T, S) = \{s_4\}$$

$$\mathrm{pre}(T, S) = \{s_1\}$$

$$\mathrm{wp}(T, S) = \{s_1, s_2, s_3\}$$

Notice that s_2 and s_4 do not have successors w.r.t. T and as a result they are included in the pre-condition of S.

In particular, we will be applying these transformers w.r.t. the transition relation $T_\Sigma(\mathbb{P})$ of a program $\mathbb{P}$, or the transition relation on states $T_{\mathbb{S}}(\mathbb{P})$, or the input-output relation $T_{\mathbb{M}}(C)$ of a statement C.

Example – Forward Analysis

For a set of traces $S \subseteq \Sigma$, let us define the *closed subset* to be

$$(\!|S|\!) = \{(t, i) \in S \mid \forall j \geq 0.\ (t, j) \in S\}$$

That is, $(\!|S|\!)$ is the largest subset of S closed under shifting the position.

As an example, we can now define an analysis that, for a program $\mathbb{P}$, produces the set of its executions $[\![\mathbb{P}]\!]$. For a program $\mathbb{P}$, let *forward analysis* of $\mathbb{P}$ be the least fixed point:

$$\mathrm{lfp}_{\subseteq} \lambda X. \left(\{(t, 0) \in \Sigma \mid t_{(0)} = (\mathfrak{l}_\vdash, m) \wedge m \neq \varepsilon\} \cup \mathrm{post}(T_\Sigma(\mathbb{P}), X)\right) \tag{2.3}$$

Lemma 2.2. For a program $\mathbb{P}$, the closed subset of the forward analysis (2.3) gives the set of all program executions that start in an non-error state.

Proof Idea. Intuitively, forward analysis collects execution prefixes, i.e., such $(t, i) \in \Sigma$ that can be produced by starting in the initial location (and a non-error memory state) and making i steps through the program. Taking closed subset, keeps only the traces that also are execution postfixes: if (t, i) is in the closed subset, then for every $j > i$, (t, j) must be in the closed subset and thus be an execution prefix, i.e., (t, i) must be an execution. □

Note how the result of forward analysis is "local": for every program location, we find execution prefixes leading to that location, but we do not immediately know what execution postfixes go from that location. Then, by taking the closed subset, we get a global result and find what set of executions the result of the analysis represents. We shall note though that the notion of closed subset is typically not used in practice. We employ it to show that forward analysis in the domain of traces produces enough information to devise an interesting subset of program executions (we will later observe a similar situation for the case of non-termination analysis). Practical analyses typically work in domains of abstract program states, and when one analysis does not produce the desired property directly, intersection of analyses is used [CC99; Arn+06].

2.3.2 Set-of-States Abstraction

When analysing a program, the most precise answer to the analysis question will be expressed as a set of program traces. In practice, we may not actually need this most precise answer. For example, if we wish to know if there are any erroneous program executions, it may be enough to find the set of reachable program states and then see whether it contains error states[13]. We will observe a similar situation with non-termination analysis. For our purposes, it will be enough to know which states can or must be visited by non-terminating program executions.

Thus the property of interest can often be expressed as a set of program states that is visited by traces of a certain form. Examples of such interesting sets are: the set of reachable states is visited by execution prefixes, a recurrent set (which will be discussed

[13] In case an error state is reachable, it may still be desirable to find at least one erroneous execution (that reaches error state), as a counterexample to safety.

later) is visited by some non-terminating execution postfixes, etc. In this case, when building an abstract domain, the first step is usually to apply *set-of-states abstraction*.

For a set of traces $S \subseteq \Sigma$, the set-of-states abstraction $\alpha_{\mathfrak{s}}(S) \in \mathbb{S}$ collects current program states of every trace:

$$\alpha_{\mathfrak{s}}(S) = \{s' \in \mathbb{S} \mid \exists (t, i) \in S.\ t_{(i)} = s'\}$$

The corresponding concretization $\gamma_{\mathfrak{s}}$, for a set of program states, $S' \subseteq \mathbb{S}$ produces the set of traces that have an element of S' at the current position

$$\gamma_{\mathfrak{s}}(S') = \{(t, i) \in \Sigma \mid t_{(i)} \in S'\}$$

For $S' \subseteq \mathbb{S}$,

$$(\!|\gamma_{\mathfrak{s}}(S')|\!) = \{(t, i) \in \Sigma \mid \forall j \geq 0.\ t_{(j)} \in S'\}$$

This way we can characterize the set of traces that only visit program states from S'.

Lemma 2.3. Set-of-states abstraction forms a Galois connection between the domains $\mathscr{P}(\Sigma)$ and $\mathscr{P}(\mathbb{S})$:

$$\mathscr{P}(\Sigma) \underset{\alpha_{\mathfrak{s}}}{\overset{\gamma_{\mathfrak{s}}}{\leftrightarrows}} \mathscr{P}(\mathbb{S})$$

Proof. This directly follows from definitions of $\alpha_{\mathfrak{s}}$ and $\gamma_{\mathfrak{s}}$. □

For example, note that the set-of-states abstraction of the forward analysis (2.3) is isomorphic to the standard notion of the *forward collecting semantics* [CC92] as it gives the set of program states that are visited by at least one execution prefix. Though, collecting semantics is often partitioned with locations and is a function $\mathbb{L} \to \mathscr{P}(\mathbb{M})$ instead of being a plain subset of $\mathbb{S}$.

When we are interested in a property that is expressed as a set of program states visited by traces of a certain form, it is often possible to characterise it directly in the domain of (sets of) program states $\mathscr{P}(\mathbb{S})$, without loss of information. For least fixed point characterisations, this is because of exact fixed point abstraction. For an interesting case of a greatest fixed point in Section 2.4, we will be able to show this as well.

Further Abstraction

Fixed points in the domain of program states are usually still not computable. Practical analyses compose set-of-states abstraction with some further abstraction. An analysis would usually introduce memory abstract domain $\mathbb{D}_\mathfrak{m}$ where elements of $\mathbb{D}_\mathfrak{m}$ represent sets of memory states (we briefly survey some memory abstract domains in Appendix 2.B). Then, the analysis would either approximate the fixed points in the domain $\mathbb{L} \to \mathbb{D}_\mathfrak{m}$ (where elements represent sets of program states) or sometimes work directly in $\mathbb{D}_\mathfrak{m}$ and make program locations implicit. We use the latter approach ourselves in Chapters 4 and 6; some other analyses [Cou+05] use it as well.

Some modern analyses use the technique of *trace partitioning* [RM07]. This implies using an abstract domain (still constructed from $\mathbb{D}_\mathfrak{m}$) where an element not only represents a set of possible current program states but also contains some information about previous or future states (usually – just about locations) of a trace. An elements of such domain represents a sets of traces directly, bypassing set-of-states abstraction. We use trace partitioning in Chapter 3.

Example 2.7. Let us look again at the program in Fig. 2.5, now from the point of view of an analysis in the domain of traces. We already considered this program in Example 2.4, but let us now give it a more rigorous treatment. In this example we will assume that the set of memory states is $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$, where $\mathbb{V} = \{x\}$ (that is, a non-error memory state maps x to an integer value).

First, recall the fixed point in (2.3) that defines forward analysis. Let us define the set I to be

$$I = \left\{(t,0) \in \Sigma \mid t_{(0)} = (l_\vdash, m) \wedge m \neq \varepsilon\right\} = \left\{(\langle s_0, s_1, \cdots\rangle, 0) \mid s_0 = (l_1, x \mapsto n \in \mathbb{Z})\right\}$$

This is the set of traces where the current position is in the beginning of the trace and points to some possible initial program state. Then, the fixed point in (2.3) can be written as

$$\mathrm{lfp}_{\subseteq} \lambda X. I \cup \mathrm{post}(T_\Sigma(\mathbb{P}), X)$$

Using Kleene fixed point theorem and properties of the post-condition, this can be written as an infinite union

$$= I \cup \mathrm{post}(T_\Sigma(\mathbb{P}), I) \cup \mathrm{post}(T_\Sigma(\mathbb{P}), \mathrm{post}(T_\Sigma(\mathbb{P}), I)) \cup \cdots = I \cup \bigcup_{k=1}^{\infty} \mathrm{post}^k(T_\Sigma(\mathbb{P}), X)$$

Let us observe a few disjuncts of this union. We have already seen I. Now,

$$\mathrm{post}(T_\Sigma(\mathbb{P}), I) = \{(\langle s_0, s_1, s_2, \cdots \rangle, 1) \mid s_0 = (l_1, x \mapsto n \in \mathbb{Z}) \wedge s_1 = (l_2, x \mapsto 0)\}$$

This is the set of traces where the current position is 1, the current position is at location l_2 and has $x = 0$ (i.e., it is the result of executing the statement at line 1), and the state at position 0 is some initial state. In a similar way,

$$\begin{aligned}\mathrm{post}(T_\Sigma(\mathbb{P}), I) = \{(\langle s_0, s_1, s_2, \cdots \rangle, 2) \mid s_0 = (l_1, x \mapsto n \in \mathbb{Z}) \wedge s_1 = (l_2, x \mapsto 0) \wedge s_2 = (l_3, x \mapsto 0)\} \\ \cup \{(\langle s_0, s_1, s_2, \cdots \rangle, 2) \mid s_0 = (l_1, x \mapsto n \in \mathbb{Z}) \wedge s_1 = (l_2, x \mapsto 0) \wedge s_2 = (l_4, x \mapsto 0)\}\end{aligned}$$

In this set of traces, where the current position is 2, and the current state is the result of either following the branch (l_2, l_3) or (l_2, l_4).

By continuing this way, in the limit, we get that the fixed point denotes the set of all execution prefixes of the program.

Now suppose that we are only interested in the *set of program states* that are visited by the execution prefixes of the program. One way to *characterize* this set is to apply the set-of-states abstraction to the original fixed point. Thus, we can take

$$\begin{aligned}&\alpha_{\mathfrak{s}}(\mathrm{lfp}_{\subseteq} \lambda X. I \cup \mathrm{post}(T_\Sigma(\mathbb{P}), X)) \\ &= \alpha_{\mathfrak{s}}(I) \cup \alpha_{\mathfrak{s}}(\mathrm{post}(T_\Sigma(\mathbb{P}), I)) \cup \alpha_{\mathfrak{s}}(\mathrm{post}(T_\Sigma(\mathbb{P}), \mathrm{post}(T_\Sigma(\mathbb{P}), I))) \cup \cdots \\ &= \{(l_1, x \mapsto n) \mid n \in \mathbb{Z}\} \cup \{(l_2, x \mapsto 0)\} \cup \{(l_3, x \mapsto 0)\} \cup \{(l_4, x \mapsto 0)\} \\ &\quad \cup \{(l_2, x \mapsto 1)\} \cup \{(l_3, x \mapsto 1)\} \cup \{(l_4, x \mapsto 1)\} \cup \cdots \\ &= \{(l_1, m_0 \in \mathbb{M})\} \cup \{(l, x \mapsto n) \mid l \in \{l_2, l_3, l_4\} \wedge n \geq 0\}\end{aligned}$$

This way, we conclude that at location l_1, x can take an arbitrary value, but at locations l_2, l_3, and l_4, x will be non-negative.

On the other hand, to produce this set of states, we do not actually have to construct a clumsy fixed point in the domain of traces. Instead, we can construct a fixed point directly in the domain of program states. First, let us observe that for the program in

Fig. 2.5, the transition relation on program states is

$$\begin{aligned}
T_{\mathbb{S}}(\mathbb{P}) = \{((l_1, n), (l_2, x \mapsto 0)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_2, x \mapsto n), (l_3, x \mapsto n)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_2, x \mapsto n), (l_4, x \mapsto n)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_3, x \mapsto n), (l_2, x \mapsto n+1)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_4, x \mapsto n), (l_4, x \mapsto n)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_1, \varepsilon), (l_2, \varepsilon))\} \cup \{((l_2, \varepsilon), (l_3, \varepsilon))\} \cup \{((l_2, \varepsilon), (l_4, \varepsilon))\} \\
\cup \{((l_3, \varepsilon), (l_2, \varepsilon))\} \cup \{((l_4, \varepsilon), (l_4, \varepsilon))\}
\end{aligned}$$

Then, let I' be the set-of-states abstraction of I, i.e., the set of initial program states:

$$I' = \alpha_{\mathfrak{s}}(I) = \{(l_1, x \mapsto n \in \mathbb{Z})\}$$

Finally, let us construct the fixed point in the domain of states:

$$\begin{aligned}
&\mathrm{lfp}_{\subseteq} \lambda X. I' \cup \mathrm{post}(T_{\mathbb{S}}(\mathbb{P}), X) \\
&= I' \cup \mathrm{post}(T_{\mathbb{S}}(\mathbb{P}), I) \cup \mathrm{post}(T_{\mathbb{S}}(\mathbb{P}), \mathrm{post}(T_{\mathbb{S}}(\mathbb{P}), I)) \cup \cdots \\
&= \{(l_1, x \mapsto n) \mid n \in \mathbb{Z}\} \cup \{(l_2, x \mapsto 0)\} \cup \{(l_3, x \mapsto 0)\} \cup \{(l_4, x \mapsto 0)\} \\
&\quad \cup \{(l_2, x \mapsto 1)\} \cup \{(l_3, x \mapsto 1)\} \cup \{(l_4, x \mapsto 1)\} \cup \cdots \\
&= \{(l_1, m_0 \in \mathbb{M})\} \cup \{(l, x \mapsto n) \mid l \in \{l_2, l_3, l_4\} \wedge n \geq 0\}
\end{aligned}$$

We obtain the same result as before, and this is not a coincidence, but the result of exact fixed point abstraction.

From this point we could perform further abstraction to produce a computable analysis. For example, we could use intervals to represent possible values of x, similarly to how we did it in Example 2.5.

2.3.3 Error State in the Abstract Domain

It is often the case that concrete error and non-error memory states are described by mathematical objects with different structure. That is, the error memory state ε is often added to the set of memory states $\mathbb{M}$ as a special element. For example, in a numeric

program with the set of integer variables $\mathbb{V}$, a non-error memory state can be described by a map from variables to integer numbers (i.e., as an element of $\mathbb{V} \to \mathbb{Z}$); but the error memory state may be a distinct object ε. Thus, the set of memory states may be $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$.

For the abstract memory domains, it is often the case that standard domains are designed to represent sets of non-error memory states. For example, for a numeric program with the set of variables $\mathbb{V}$, there is a standard way to construct the domain of polyhedra (i.e., conjunctions of linear inequalities) over $\mathbb{V}$ (which we will denote by $\mathbb{D}_{\text{poly}}(\mathbb{V})$), but there is no single standard way to represent the error memory state, and we would need to make an additional construction over $\mathbb{D}_{\text{poly}}(\mathbb{V})$. Arguably the simplest construction introduces an artificial top element and takes $\mathbb{D}_{\mathfrak{m}} = \mathbb{D}_{\text{poly}}(\mathbb{V}) \cup \{\top_{\mathfrak{m}}\}$, where for every $d \in \mathbb{D}_{\text{poly}}(\mathbb{V})$, $d \sqsubseteq_{\mathfrak{m}} \top_{\mathfrak{m}}$. The concretization of the new top element $\top_{\mathfrak{m}}$ is the whole set of memory states: $\gamma_{\mathfrak{m}}(\top_{\mathfrak{m}}) = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$. For the other elements, concretization is the standard concretization of the polyhedral domain. There exist other constructions, but we will not discuss them in this work, as the simple construction is sufficient for our purposes.

For some analyses, it may actually not be necessary to represent errors in the abstract domain. It may be the case that an analysis by definition approximates some set of non-error states, and in all computation steps all abstract states are guaranteed to be error-free. A notable example is backward analysis (based on computation of pre-condition or predecessors) when it is initialized with some error-free abstract state. Then, all pre-conditions/predecessors of such abstract state are also error-free.

2.4 Non-Termination Analyses and Recurrent Sets

In this section, we finally introduce the analyses of the non-terminating behaviours of a program. We start by defining a concrete analysis that finds all the non-terminating executions of a program.

Existential Non-Termination Analysis

For a program $\mathbb{P}$, *existential non-termination* analysis of $\mathbb{P}$ is the greatest fixed point:

$$\mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{(t,i) \in \Sigma \mid t_{(i)} \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\Sigma}(\mathbb{P}), X)\big) \tag{2.4}$$

Lemma 2.4. For a program $\mathbb{P}$ the closed subset of its existential non-termination analysis (2.4) gives the set of all non-terminating semi-executions of the program.

Proof Idea. Intuitively, existential non-termination analysis retains non-terminating execution postfixes (we present the full proof in Appendix 2.A). Taking closed subset keeps only the traces that also are execution prefixes: if (t,i) is in the closed subset, then for every j, s.t. $0 \le j < i$, (t,j) must be in the closed subset and thus must be an execution postfix, i.e., (t,i) must be a semi-execution. □

Note how finding the set of non-terminating executions immediately breaks into two sub-problems. The first one is to find the set of non-terminating execution postfixes, i.e., to approximate the fixed point (2.4). The second one is to produce the set of non-terminating executions from the set of non-terminating execution postfixes. On the theoretical level, one can intersect the closed subset of existential non-terminating analysis with $\{(t,i) \in \Sigma \mid \mathfrak{p}(t)_{(0)} = \mathfrak{l}_{\vdash}\}$ (i.e. keep only the traces that start in the initial location). A more practical way is to intersect existential non-termination analysis with forward analysis.

In practice, the two sub-problems are solved by different techniques, and in this work, we focus on the first subproblem and mostly ignore the second one[14].

Existential Recurrent Set

For a program $\mathbb{P}$, a set of program states $S_{\exists} \subseteq \mathbb{S}$ is an *existential recurrent set* iff for every $s \in S_{\exists}$, s is non-error and non-final and there exists $s' \in S_{\exists}$, s.t. $(s,s') \in T_{\mathbb{S}}(\mathbb{P})$. In other words, this is a set $S_{\exists}$ of non-error and non-final states, s.t. once the program reaches some $s \in S_{\exists}$, it *may* choose (as it takes its non-deterministic choices) to stay

[14] Reachability can be proved by a specialized tool, e.g., a bounded model checker or a model checker based on abstraction refinement. Such model checkers can produce a genuine execution reaching a specific interesting set of states.

in $S_\exists$ forever. Note that by definition, an empty set is trivially existentially recurrent. In the literature (e.g., [Che+14]) a similar but stronger notion of an *open recurrent set* is often used, that requires all the program states in the open recurrent set to be reachable from some initial program state.

Lemma 2.5. The largest existential recurrent set can be characterized as

$$\mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X)\big)$$

Proof Idea. The proof is by definitions of predecessor transformer and universal recurrent set. We present it in detail in Appendix 2.A. □

Example 2.8. Let us now construct the existential recurrent set for the program in Fig. 2.5. First, let us recall that the transition relation of the program is

$$\begin{aligned}
T_{\mathbb{S}}(\mathbb{P}) = \{((l_1, n), (l_2, x \mapsto 0)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_2, x \mapsto n), (l_3, x \mapsto n)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_2, x \mapsto n), (l_4, x \mapsto n)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_3, x \mapsto n), (l_2, x \mapsto n+1)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_4, x \mapsto n), (l_4, x \mapsto n)) \mid n \in \mathbb{Z}\} \\
\cup \{((l_1, \varepsilon), (l_2, \varepsilon))\} \cup \{((l_2, \varepsilon), (l_3, \varepsilon))\} \cup \{((l_2, \varepsilon), (l_4, \varepsilon))\} \\
\cup \{((l_3, \varepsilon), (l_2, \varepsilon))\} \cup \{((l_4, \varepsilon), (l_4, \varepsilon))\}
\end{aligned}$$

Then, let I be the set

$$\begin{aligned}
I &= \{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \\
&= \{(l, x \mapsto n) \mid l \in \{l_1, l_2, l_3\} \wedge n \in \mathbb{Z}\}
\end{aligned}$$

Then, the existential recurrent set can be characterized as:

$$R_\exists = \mathrm{gfp}_{\subseteq}\, \lambda X. I \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X)$$

Via Kleene fixed point theorem, this can be written as a countable intersection $\bigcap_{i=0}^{\infty} R_i$,

where

$$R_0 = I \cap \text{pre}(T_{\mathbb{S}}(\mathbb{P}), \mathbb{S}) = I$$

(This is because $\text{pre}(T_{\mathbb{S}}(\mathbb{P}), \mathbb{S})$ is the set of states that have at leastt one successor via $T_{\mathbb{S}}(\mathbb{P})$, and in this case this is the set of all states $\mathbb{S}$)

$$R_1 = I \cap \text{pre}(T_{\mathbb{S}}(\mathbb{P}), R_0) = \{(l_1, x \mapsto 0)\} \cup \{(l_2, x \mapsto n \in \mathbb{Z})\} \cup \{(l_3, x \mapsto n \in \mathbb{Z})\}$$

$$R_2 = I \cap \text{pre}(T_{\mathbb{S}}(\mathbb{P}), R_1) = R_1$$

Thus, the recurrent set is

$$R_{\exists} = \{(l_1, x \mapsto 0)\} \cup \{(l_2, x \mapsto n \in \mathbb{Z})\} \cup \{(l_3, x \mapsto n \in \mathbb{Z})\}$$

Note that not all the states in the recurrent set are reachable in an execution. In particular, in an execution, only non-negative values of x are reachable, while the recurrent set allows x to be negative at locations l_2 and l_3. This happens because the recurrent set is the set-of-states abstraction of the set of non-terminating execution *postfixes* (rather than executions)[15]. Indeed, one can see that it is possible to build a non-terminating execution postfix by starting in a state, where x is negative, e.g., $(l_2, x \mapsto -1)$.

We can actually show that existential recurrent set is exact fixed point abstraction of existential non-termination analysis.

Lemma 2.6. The largest existential recurrent set is set-of-states abstraction of existential non-termination analysis.

Proof. Intuitively the largest existential recurrent set $R_{\exists}$ the largest set of states, s.t. from every element of $R_{\exists}$ we can start a non-terminating semi-execution that only visits elements of $R_{\exists}$.

Non-termination analysis produces the set $R'_{\exists}$ of all non-terminating execution postfixes, and by applying set-of-states abstraction to it, we produce the set of all program states from which we can start a non-terminating execution postfix. Moreover,

[15] In principle, we could define a "reachable existential recurrent" set to only consist of reachable states, along the lines of

$$R^{\text{reach}}_{\exists} = \text{gfp}_{\subseteq} \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final and reachable}\} \cap \text{pre}(T_{\mathbb{S}}(\mathbb{P}), X)\big)$$

We do not do this for practical reasons. In Chapters 3 and 4 we will build a procedures that can compute under-approximations of our original notion of recurrent set, but not of this "reachable recurrent set".

the set of non-terminating execution postfixes is closed under shifting the current position forward, and non-terminating execution postfixes in $R'_\exists$ only visit the states from $\alpha_\mathfrak{s}(R'_\exists)$.

That is, by applying set-of-states abstraction to $R'_\exists$, we produce the largest set of all program states from which we can start a non-terminating execution postfix, i.e., the largest existential recurrent set. □

The transition from sets of non-terminating execution to recurrent sets is done for a number of reasons. On one hand, modern abstract domains can only efficiently represent sets of program states (e.g., as elements of $\mathbb{L} \to \mathbb{D}_\mathfrak{m}$), and not sets of traces. Even trace partitioning domains maintain very little information on top of the set of possible current program states. On the other hand, depending on the goals of the analysis, knowing just the recurrent set may be enough. For example, if we wanted to prove that a program has at least one non-terminating execution, we would need to:

(i) find an *under-approximation* of an existential recurrent set; we develop techniques for that in Chapters 3 and 4.

(ii) find at least one program state in the existential recurrent set that is definitely reachable from some initial state; there exist techniques for that, including for non-numeric programs [Ber+13]. We do not develop or discuss these techniques in this work.

This certifies the existence of at least one non-terminating execution. If needed, a non-terminating execution postfix may be produced from the existential recurrent set by starting in some program state (reachable from some initial program state) and repeatedly picking a successor that is both in the post-condition of the current state and in the recurrent set. A prefix of the non-terminating execution may be produced by the employed reachability analysis.

Universal Recurrent Set

Now that we defined an *existential* recurrent set, it is conceivable that there exists a notion of a recurrent set that uses the different quantifier.

For a program $\mathbb{P}$, let a *universal recurrent set* be a set of states $S_\forall \subseteq \mathbb{S}$, s.t. for every $s \in S_\forall$, s is non-error and non-final, and for every $s' \in \mathbb{S}$, s.t. $(s, s') \in T_\mathbb{S}(\mathbb{P})$, $s' \in S_\forall$. Intu-

itively, this is a set of non-error and non-final states that, once reached by a program, cannot be escaped. Again, in literature (e.g., [Che+14]) a similar but stronger notion of a *closed recurrent set* is often used, that requires all the program states in the closed recurrent set to be reachable from some initial program state.

From the point of view of proving non-termination, the notion of universal recurrent set is useful in that under certain conditions, a universal recurrent set is also an existential recurrent set and certifies the existence of a non-termination execution postfix. At the same time, it might be easier to develop a technique that finds universal recurrent sets. This is the topic of Chapter 4.

In this work, we do not use the counterpart for the universal recurrent set in the domain of traces, and prefer to characterize the largest universal recurrent set directly.

Lemma 2.7. The largest universal recurrent set can be characterized as

$$\mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), X)\big)$$

Proof Idea. The proof is by definitions of pre-condition and universal recurrent set. We present it in detail in Appendix 2.A. □

In principle, it is possible to define the counterpart of universal recurrent set in the domain of traces along the following lines. Universal recurrent set on traces is a set $R'_\forall \subseteq \Sigma$, s.t.

(i) every trace in $R'_\forall$ is a non-terminating execution postfix;

(ii) for every $(t, i) \in R'_\forall$, for every trace t', s.t. for $j = 0 \cdots i$, $t'_{(j)} = t_{(j)}$, $t' \in R'_\forall$.

We will not use this definition and we will not give a fixed point characterisation for the universal recurrent set on traces anywhere in this work.

2.5 Structured Programs

The notion of a program introduced in Section 2.2 is quite general and is often *too* general, as many algorithms can be expressed with programs of restricted form. On the other hand, restricted programs may be easier to analyse. In this section, we define

a restricted class of *structured programs*. Later, in Chapters 4 and 6 we will develop analyses that are specific to structured programs.

We define the language of *structured programs* $\mathbb{C}$. Given a set of atomic statements $\mathbb{A}$, elements of the language are built as follows:

$C ::=$	$a \in \mathbb{A}$	atomic statement
	$C_1 \,;\, C_2$	sequential composition: executes C_1 and then C_2
$\mid$	$C_1 + C_2$	branch: non-deterministically branches to either C_1 or C_2
$\mid$	C^*	loop: repeats C a non-deterministic (possibly 0) number of times.

For the sake of uniformity, we will usually call the elements of $\mathbb{C}$ *statements* rather than programs, distinguishing between atomic and compound statements.

Note that the basic branching and looping statements are non-deterministic, but we can still express standard deterministic constructs. Standard conditional statement "`if`(θ) C_1 `else` C_2" can be expressed as $([\theta] \,;\, C_1) + ([\neg\theta] \,;\, C_2)$, and standard loop "`while`$(\theta)$ C" can be expressed as $([\theta] \,;\, C)^* \,;\, [\neg\theta]$. That is, the language of structured programs can be seen as a simple imperative programming language[16].

Most examples presented in this work can be represented as structured programs[17]. For example, the program in Fig. 2.5 can be written as $x \leftarrow 0 \,;\, (x \leftarrow x+1)^*$. The program in Fig. 2.2a can be written as

$$x \leftarrow 4 \,;\, y \leftarrow -1 \,;\, ([x \geq 0] \,;\, x \leftarrow x + y)^* \,;\, [x < 0]$$

Input-Output relation of a Structured Program

For structured programs, we will often not be explicitly interested in the set of their executions, as there is another way to describe their behaviour. Namely, we lift the notion of input-output relation to compound statements (for atomic statements, we

[16] Or rather a modelling language, as imperative programming languages usually do not admit assumption statements. The effect of an assumption statement is to discard (declare as a non-execution) the whole trace when the current program state (from which the assumption statement is executed) does not satisfy the assumption condition. This is usually not possible in an imperative programming language.

[17] On the other hand, programs that (in their pseudocode representation) include statements affecting the control flow (`break`, `return`, `continue`, `goto`, etc) cannot immediately be written as structured programs. An example of such program can be found in Fig 4.14 of Example 4.5. If needed, we can rewrite such a program into an equivalent structured one (this is what we do in that example).

assume that their input-output relation are given). For $C, C_1, C_2 \in \mathbb{C}$,

$$\begin{aligned} T_{\mathbb{M}}(C_1\,;\,C_2) &= T_{\mathbb{M}}(C_2) \circ T_{\mathbb{M}}(C_1) \\ T_{\mathbb{M}}(C_1 + C_2) &= T_{\mathbb{M}}(C_1) \cup T_{\mathbb{M}}(C_2) \\ T_{\mathbb{M}}(C^*) &= \mathrm{lfp}_{\subseteq} \lambda X.\Delta_{\mathbb{M}} \cup (X \circ T_{\mathbb{M}}(C)) \end{aligned} \tag{2.5}$$

For structured programs, the input-output relation has the same meaning as for atomic statements. If for $C \in \mathbb{C}$, $(m_1, m_2) \in T_{\mathbb{M}}(C)$, then if we start executing C in the memory state m_1, C may terminate[18] in memory state m_2.

Post-Condition of a Statement

For a compound statement, a post-condition via its input-output relation can also be computed by induction over the statement structure. $C, C_1, C_2 \in \mathbb{C}$ and $M \subseteq \mathbb{M}$:

$$\begin{aligned} \mathrm{post}(T_{\mathbb{M}}(C_1\,;\,C_2), M) &= \mathrm{post}(T_{\mathbb{M}}(C_2), \mathrm{post}(T_{\mathbb{M}}(C_1), M)) \\ \mathrm{post}(T_{\mathbb{M}}(C_1 + C_2), M) &= \mathrm{post}(T_{\mathbb{M}}(C_1), M) \cup \mathrm{post}(T_{\mathbb{M}}(C_2), M) \\ \mathrm{post}(T_{\mathbb{M}}(C^*), M) &= \mathrm{lfp}_{\subseteq} \lambda X.M \cup \mathrm{post}(T_{\mathbb{M}}(C), X) \end{aligned} \tag{2.6}$$

For predecessors and pre-condition transformers, the construction is similar.

This is useful when performing abstract computation in an analysis. Let us assume we are given the *memory abstract domain* $\mathbb{D}_{\mathfrak{m}}$, with least element $\bot_{\mathfrak{m}}$, greatest element $\top_{\mathfrak{m}}$, partial order $\sqsubseteq_{\mathfrak{m}}$, join $\sqcup_{\mathfrak{m}}$ and concretization $\gamma_{\mathfrak{m}}$. Usually, the transformer $\mathrm{post}^{\mathfrak{m}}$ is only given for atomic statements. Using (2.6), we can lift it to compound statements:

$$\begin{aligned} \mathrm{post}^{\mathfrak{m}}(C_1\,;\,C_2, a) &= \mathrm{post}^{\mathfrak{m}}(C_2, \mathrm{post}^{\mathfrak{m}}(C_1, a)) \\ \mathrm{post}^{\mathfrak{m}}(C_1 + C_2, a) &= \mathrm{post}^{\mathfrak{m}}(C_1, a) \sqcup_{\mathfrak{m}} \mathrm{post}^{\mathfrak{m}}(C_2, a) \\ \mathrm{post}^{\mathfrak{m}}(C^*, a) &\quad \text{is the stable limit of the chain } \{a'_i\}_{i\geq 1}, \text{ where} \\ &\qquad a'_1 = a \\ &\qquad a'_i = a'_{i-1} \triangledown_{\mathfrak{m}} (a'_{i-1} \sqcup_{\mathfrak{m}} \mathrm{post}^{\mathfrak{m}}(C, a'_{i-1})), \text{ for } i > 1 \end{aligned} \tag{2.7}$$

[18] We say "may terminate" due to possible non-determinism and existence of erroneous and non-terminating behaviours. For example, if there exists $m_3 \neq m_2$, s.t. $(m_1, m_3) \in T_{\mathbb{M}}(C)$, executing C from m_1 might produce m_3 instead of m_2, depending on non-deterministic choices. Similarly, if additionally $(m_1, \varepsilon) \in T_{\mathbb{M}}(C)$, executing C from m_1 might fail instead of producing m_2. Also, even when $(m_1, m_2) \in T_{\mathbb{M}}(C)$ the execution of a compound (not of an atomic though) statement C from m_1 might not terminate.

For predecessors and pre-condition transformers, the construction is similar. This means that we can always assume that we can compute the transformers for arbitrary (both atomic and compound) statements. For example, in Chapters 4 and 6, we will be computing post-conditions of whole loop bodies, disregarding their internal structure.

Graph of a Structured Program

Structured programs can be seen as a compact way of expressing programs of a certain form. For a structured program $C \in \mathbb{C}$, we can build the corresponding unstructured program $\mathbb{P}(C)$, which we call the *graph of* C. When talking about an execution of a structured program, we will actually mean an execution of its graph.

The graph is constructed in the following way.

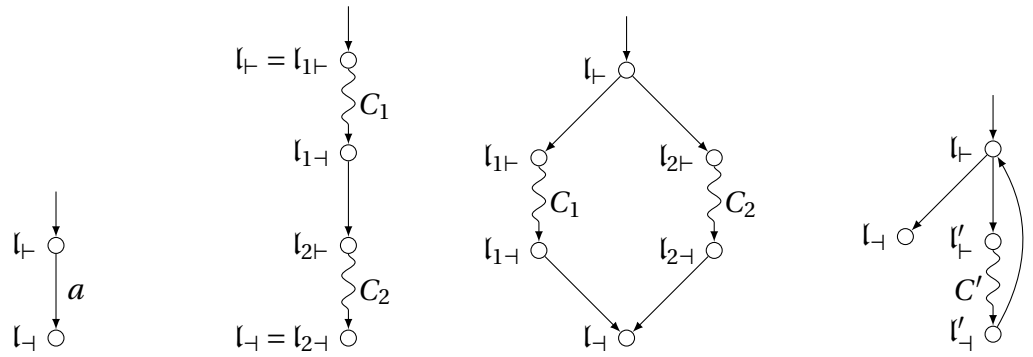
(i) For an atomic statement $a \in \mathbb{A}$, $\mathbb{P}(a)$ is a graph $(\mathbb{L}, \ell_\vdash, \mathbb{E}, \mathfrak{c})$ where $\mathbb{L} = \{\ell_\vdash, \ell_\dashv\}$ – a pair of fresh program locations; $\mathbb{E} = \{(\ell_\vdash, \ell_\dashv)\}$; and $\mathfrak{c} = \{(\ell_\vdash, \ell_\dashv) \mapsto a\}$. It is actually the case that for *every* statement C, $\mathbb{P}(C)$ will have a single final location that we will always denote by $\ell_\dashv$.

(ii) For the sequential composition $C = C_1 \,;\, C_2$, let $\mathbb{P}(C_1) = (\mathbb{L}_1, \ell_{1\vdash}, \mathbb{E}_1, \mathfrak{c}_1)$ with final location $\ell_{1\dashv}$. Let $\mathbb{P}(C_2) = (\mathbb{L}_2, \ell_{2\vdash}, \mathbb{E}_2, \mathfrak{c}_2)$ with final location $\ell_{2\dashv}$. Without loss of generality let us also assume that the program locations of $\mathbb{P}(C_1)$ and $\mathbb{P}(C_2)$ are disjoint: $\mathbb{L}_1 \cap \mathbb{L}_2 = \varnothing$. Then $\mathbb{P}(C) = (\mathbb{L}, \ell_\vdash, \mathbb{E}, \mathfrak{c})$ with final location $\ell_\dashv$ where $\mathbb{L} = \mathbb{L}_1 \cup \mathbb{L}_2$; $\ell_\vdash = \ell_{1\vdash}$; $\ell_\dashv = \ell_{2\dashv}$; $\mathbb{E} = \mathbb{E}_1 \cup \mathbb{E}_2 \cup \{(\ell_{1\dashv}, \ell_{2\vdash})\}$; $\mathfrak{c} = \mathfrak{c}_1 \cup \mathfrak{c}_2 \cup \{(\ell_{1\dashv}, \ell_{2\vdash}) \mapsto \texttt{skip}\}$.

(iii) For the branching $C = C_1 + C_2$, let $\mathbb{P}(C_1) = (\mathbb{L}_1, \ell_{1\vdash}, \mathbb{E}_1, \mathfrak{c}_1)$ with final location $\ell_{1\dashv}$; let $\mathbb{P}(C_2) = (\mathbb{L}_2, \ell_{2\vdash}, \mathbb{E}_2, \mathfrak{c}_2)$ with final location $\ell_{2\dashv}$; let $\mathbb{L}_1 \cap \mathbb{L}_2 = \varnothing$; and let $\ell_\vdash, \ell_\dashv \notin \mathbb{L}_1 \cup \mathbb{L}_2$. Then $\mathbb{P}(C) = (\mathbb{L}, \ell_\vdash, \mathbb{E}, \mathfrak{c})$ with final location $\ell_\dashv$ where $\mathbb{L} = \mathbb{L}_1 \cup \mathbb{L}_2 \cup \{\ell_\vdash, \ell_\dashv\}$; $\mathbb{E} = \mathbb{E}_1 \cup \mathbb{E}_2 \cup \{(\ell_\vdash, \ell_{1\vdash}), (\ell_\vdash, \ell_{2\vdash}), (\ell_{1\dashv}, \ell_\dashv), (\ell_{1\dashv}, \ell_\dashv)\}$; $\mathfrak{c} = \mathfrak{c}_1 \cup \mathfrak{c}_2 \cup \{(\ell_\vdash, \ell_{1\vdash}) \mapsto \texttt{skip}, (\ell_\vdash, \ell_{2\vdash}) \mapsto \texttt{skip}, (\ell_{1\dashv}, \ell_\dashv) \mapsto \texttt{skip}, (\ell_{1\dashv}, \ell_\dashv) \mapsto \texttt{skip}\}$.

(iv) For the loop $C = C'^*$, let $\mathbb{P}(C') = (\mathbb{L}', \ell'_\vdash, \mathbb{E}', \mathfrak{c}')$ with final location $\ell'_\dashv$ and let $\ell_\vdash, \ell_\dashv \notin \mathbb{L}'$. Then $\mathbb{P}(C) = (\mathbb{L}, \ell_\vdash, \mathbb{E}, \mathfrak{c})$ with final location $\ell_\dashv$ where $\mathbb{L} = \mathbb{L}' \cup \{\ell_\vdash, \ell_\dashv\}$; $\mathbb{E} = \mathbb{E}' \cup \{(\ell_\vdash, \ell'_\vdash), (\ell'_\dashv, \ell_\vdash), (\ell_\vdash, \ell_\dashv)\}$; $\mathfrak{c} = \mathfrak{c}' \cup \{(\ell_\vdash, \ell'_\vdash) \mapsto \texttt{skip}, (\ell'_\dashv, \ell_\vdash) \mapsto \texttt{skip}, (\ell_\vdash, \ell_\dashv) \mapsto \texttt{skip}\}$.

These constructions are demonstrated in Fig 2.8. Note that resulting programs may have redundant locations and skip-edges, but for simplicity we do not attempt to minimize the programs.



(a) Atomic. (b) Sequence. (c) Branching. (d) Loop.

Figure 2.8: Unstructured programs corresponding to structured programs.

The Correspondence Formally Defined

The following lemma formally defines correspondence between a structured program $C \in \mathbb{C}$ and its unstructured counterpart $\mathbb{P}(C)$.

Lemma 2.8. For a structured program $C \in \mathbb{C}$ and its graph $\mathbb{P}(C)$ with an initial location $\ell_\vdash$ and a final location $\ell_\dashv$, a pair of memory states $(m, m') \in T_{\mathbb{M}}(C)$ iff there exists a terminating execution $\tau \in [\![\mathbb{P}(C)]\!]$, s.t. $\tau_{(0)} = (\ell_\vdash, m)$ and $\tau_{(\dashv)} = (\ell_\dashv, m')$ (i.e., $\tau = \langle (\ell_\vdash, m), \cdots, (\ell_\dashv, m')^{\mathbb{N}} \rangle$). Informally, we can say that the input-output relation of C summarizes the set of terminating executions of $\mathbb{P}(C)$.

Proof idea. The proof is by induction over the structure of a statement and by direct application of the definition of an execution and an input-output relation of a non-atomic statement (2.5). We present the more detailed proof in Appendix 2.A. □

2.6 Related Work

In [CC12], a different notion of trace semantics is introduced (a notable difference being that it separates current program state from the sequence of states visited by the execution) that also allows to define analyses for termination and non-termination. In [RM07], a simpler notion of semantics is introduces, allowing to formally define trace partitioning for forward analysis.

2.7 Chapter Conclusion

In this chapter, we have formally introduced programs without procedures and their semantics which are sets of executions. These notions (especially the notion of an execution) might be considered non-standard, but they allow us to perform further theoretical exercises, without which this work would not look complete. First, we are able to explain the connection between practical analyses based on sets of states and different subsets of the executions of a program (in particular – between a recurrent set and a set of non-terminating executions). Second, later in Chapter 3, we will be able to formally introduce trace partitioning for backward analysis, which to our knowledge has not been done before.

2.A Omitted Proofs

Lemma 2.1. Let $\mathscr{L}_\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D}_\sharp$. Also, let the concrete transfer function $F_\flat : \mathscr{L}_\flat \to \mathscr{L}_\flat$ and its abstract approximation $F_\sharp : \mathbb{D}_\sharp \to \mathbb{D}_\sharp$ be monotone and such that $\alpha \circ F_\flat = F_\sharp \circ \alpha$. Then

$$\alpha(\mathrm{gfp}_{\sqsubseteq_\flat} F_\flat) \sqsubseteq_\sharp \mathrm{gfp}_{\sqsubseteq_\sharp} F_\sharp$$

Proof.

$$\alpha(\mathrm{gfp}_{\sqsubseteq_\flat} F_\flat)$$

Via Knaster-Tarski theorem

$$= \alpha(\bigsqcup\nolimits_\flat \{L \in \mathscr{L}_\flat \mid F_\flat(L) \sqsupseteq_\flat L\})$$

Since abstraction function preserves upper bounds… [CC79, theorem 5.3.0.5]

$$= \bigsqcup\nolimits_\sharp \{\alpha(L) \mid F_\flat(L) \sqsupseteq_\flat L\}$$

Since $F_\flat(L) \sqsupseteq_\flat L$ *implies* $\alpha(F_\flat(L)) \sqsupseteq_\sharp \alpha(L)$

$$\sqsubseteq_\sharp \bigsqcup\nolimits_\sharp \{\alpha(L) \mid \alpha(F_\flat(L)) \sqsupseteq_\sharp \alpha(L)\} = \bigsqcup\nolimits_\sharp \{\alpha(L) \mid F_\sharp(\alpha(L)) \sqsupseteq_\sharp \alpha(L)\}$$

$$= \bigsqcup\nolimits_\sharp \{D \in \mathbb{D}_\sharp \mid F_\sharp(D) \sqsupseteq_\sharp D\} = \mathrm{gfp}_{\sqsubseteq_\sharp} F_\sharp$$ □

Lemma 2.4. For a program $\mathbb{P}$ the closed subset of its existential non-termination analysis (2.4) gives the set of all non-terminating semi-executions of the program.

Proof. Note that existential non-termination analysis retains non-terminating execution postfixes. Indeed, consider the infinite iteration sequence $\{S_j\}_{j\geq 0}$ where

$$S_0 = \Sigma$$

$$S_j = \{(t,i) \in \Sigma \mid t_{(i)} \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_\Sigma(\mathbb{P}), S_{j-1}), \text{ for } j \geq 1$$

From Kleene fixed point theorem,

$$\bigcap \{S_j\}_{j\geq 0} = \mathrm{gfp}_{\subseteq} \lambda X.\big(\{(t,i) \in \Sigma \mid t_{(i)} \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_\Sigma(\mathbb{P}), X)\big)$$

For the j-th set S_j, and for an arbitrary trace $(t,i) \in S_j$, let us observe a subsequence of j program states starting at position i: $\langle t_{(i)}, t_{(i+1)}, \ldots, t_{(i+j-1)}\rangle$. This subsequence has the following properties (provable by induction on j and by definition of pre):

(i) for $k = 0 \dots j-1$, $t_{(i+k)}$ is non-error and non-final;

(ii) for $j \geq 1$ and $k = 0 \dots j-1$, $(t_{(i+k)}, t_{(i+k+1)}) \in T_{\mathbb{S}}(\mathbb{P})$.

Then, let us assume that some trace (t', i') is not a non-terminating execution postfix. Then at least one of the following is true.

(i) For some $k' \geq 0$, $t'_{(i'+k')}$ is an error state. Then, $(t', i') \notin S_{k'+1}$.

(ii) For some $k' \geq 0$, $t'_{(i'+k')}$ is final. Then, $(t', i') \notin S_{k'+1}$.

(iii) For some $k' \geq 0$, $(t'_{(i'+k')}, t'_{(i'+k'+1)}) \notin T_{\mathbb{S}}(\mathbb{P})$. $(t', i') \notin S_{k'+1}$.

In all three cases, $(t', i') \notin \bigcap\{S_j\}_{j\geq 0}$.

On the other hand, if (t', i') is a non-terminating execution postfix, $(t', i') \in S_j$ for every $j \geq 0$, i.e., $(t', i') \in \bigcap\{S_j\}_{j\geq 0}$.

Finally, taking closed subset keeps only the traces that also are execution prefixes: if (t, i) is in the closed subset, then for every k, s.t. $0 \leq k < i$, (t, k) must be in the closed subset and thus must be an execution postfix, i.e., (t, i) must be a semi-execution. □

Lemma 2.5. The largest existential recurrent set can be characterized as

$$\mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X)\big)$$

Proof. The largest existential recurrent set is the largest set $R_\exists \in \mathbb{S}$, s.t. every program state in $R_\exists$ is non-error and non-final and $\forall s \in R_\exists. \exists s' \in R_\exists. (s, s') \in T_{\mathbb{S}}(\mathbb{P})$. Note that the set $\{s \in \mathbb{S} \mid \exists s' \in R_\exists. (s, s') \in T_{\mathbb{S}}(\mathbb{P})\}$ is $\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists)$. That is, $R_\exists$ is the largest set of non-error non-final program states, s.t. $\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) \supseteq R_\exists$. Equivalently, $R_\exists$ is the largest set, s.t. $\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) \supseteq R_\exists$.

Indeed, if $\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) \supseteq R_\exists$ then every program state in $R_\exists$ is non-error and non-final, and from definition of predecessors, every program state in $\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists)$ is non-error and non-final. That is,

$$\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) = \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) \supseteq R_\exists$$

Conversely, if $R_\exists$ is a set of non-error non-final program states, and $\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) \supseteq R_\exists$, then every program state in $\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists)$ is non error and non-final, and hence $\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) = \{s \in \mathbb{S} \mid s \text{ is non-error and not final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), R_\exists) \supseteq R_\exists$.

Then from Knaster-Tarski theorem,

$$R_{\exists} = \mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X)\big) \qquad \square$$

Lemma 2.7. The largest universal recurrent set can be characterized as

$$\mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), X)\big)$$

Proof. The largest universal recurrent set is the largest set $R \in \mathbb{S}$, s.t. every program state in R is non-error and non-final and $\forall s \in R.\, (\forall s' \in \mathbb{S}.\, (s, s') \in T_{\mathbb{S}}(\mathbb{P}) \Rightarrow s' \in R)$. Note that the set $\{s \in \mathbb{S} \mid \forall s' \in \mathbb{S}.\, (s, s') \in T_{\mathbb{S}}(\mathbb{P}) \Rightarrow s' \in R\}$ is $\mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R)$. That is, R is the largest set of non-error non-final program states, s.t. $\mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) \supseteq R$. Equivalently, R is the largest set, s.t. $\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) \supseteq R$.

Indeed, if $\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) \supseteq R$ then every program state in R is non-error and non-final, hence every program state in $\mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R)$ is non-error and non-final, and $\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) = \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) \supseteq R$. Conversely, if R is a set of non-error non-final program states, and $\mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) \supseteq R$, then every program state in $\mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R)$ is non error and non-final, and $\mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) = \{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), R) \supseteq R$.

Then from Knaster-Tarski theorem,

$$R = \mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and non-final}\} \cap \mathrm{wp}(T_{\mathbb{S}}(\mathbb{P}), X)\big) \qquad \square$$

Lemma 2.8. For a structured program $C \in \mathbb{C}$ and its graph $\mathbb{P}(C)$ with an initial location $\mathfrak{l}_{\vdash}$ and a final location $\mathfrak{l}_{\dashv}$, a pair of memory states $(m, m') \in T_{\mathbb{M}}(C)$ iff there exists a terminating execution $\tau \in [\![\mathbb{P}(C)]\!]$, s.t. $\tau_{(0)} = (\mathfrak{l}_{\vdash}, m)$ and $\tau_{(\dashv)} = (\mathfrak{l}_{\dashv}, m')$ (i.e., $\tau = \langle (\mathfrak{l}_{\vdash}, m), \cdots, (\mathfrak{l}_{\dashv}, m')^{\mathbb{N}} \rangle$). Informally, we can say that the input-output relation of C summarizes the set of terminating executions of $\mathbb{P}(C)$.

Proof. We proceed by induction over the structure of a statement C.

(i) For an atomic statement $a \in \mathbb{A}$ (Fig. 2.8a). From the definition of execution, the executions of $\mathbb{P}(a)$ are of the form $\langle (\mathfrak{l}_{\vdash}, m), (\mathfrak{l}_{\dashv}, m')^{\mathbb{N}} \rangle$ where $(m, m') \in T_{\mathbb{M}}(a)$. That is, for $\tau \in [\![\mathbb{P}(a)]\!]$, $\tau_{(0)} = (\mathfrak{l}_{\vdash}, m)$ and $\tau_{(\dashv)} = (\mathfrak{l}_{\dashv}, m')$ iff $(m, m') \in T_{\mathbb{M}}(a)$.

(ii) For a sequential composition $C = C_1 \,;\, C_2$ (Fig. 2.8b).

Forward. If $(m, m') \in T_{\mathbb{M}}(C)$ then from (2.5) there exists $m'' \in \mathbb{M}$, s.t. $(m, m'') \in T_{\mathbb{M}}(C_1)$ and $(m'', m') \in T_{\mathbb{M}}(C_2)$. Then, from the inductive hypothesis, the terminating executions of $\mathbb{P}(C_1)$ are of the form $\langle(\mathfrak{l}_{1\vdash}, m), \cdots, (\mathfrak{l}_{1\dashv}, m'')^{\mathbb{N}}\rangle$, and the terminating executions of $\mathbb{P}(C_2)$ are of the form $\langle(\mathfrak{l}_{2\vdash}, m''), \cdots, (\mathfrak{l}_{2\dashv}, m')^{\mathbb{N}}\rangle$. That is, terminating executions of C are of the form $\langle(\mathfrak{l}_{1\vdash}, m), \cdots, (\mathfrak{l}_{1\dashv}, m''), (\mathfrak{l}_{2\vdash}, m'), \cdots, (\mathfrak{l}_{2\dashv}, m')^{\mathbb{N}}\rangle$, and for $\tau \in [\![\mathbb{P}(a)]\!]$, $\tau_{(0)} = (\mathfrak{l}_{\vdash}, m)$ and $\tau_{(\dashv)} = (\mathfrak{l}_{\dashv}, m')$.

Backward. Let us assume that the terminating executions of C are of the form $\langle(\mathfrak{l}_{1\vdash}, m), \cdots, (\mathfrak{l}_{2\dashv}, m')^{\mathbb{N}}\rangle$ for some $m, m' \in \mathbb{M}$. Every terminating execution τ must first go through the body of $\mathbb{P}(C_1)$, visit the location $\mathfrak{l}_{1\dashv}$, then $\mathfrak{l}_{2\vdash}$ and then proceed to the body of $\mathbb{P}(C_2)$. That is, there exists a memory state $m'' \in \mathbb{M}$, s.t. $\tau = \langle(\mathfrak{l}_{1\vdash}, m), \cdots, (\mathfrak{l}_{1\dashv}, m''), (\mathfrak{l}_{2\vdash}, m'), \cdots, (\mathfrak{l}_{2\dashv}, m')^{\mathbb{N}}\rangle$ and therefore the terminating executions of $\mathbb{P}(C_1)$ are of the form $\langle(\mathfrak{l}_{1\vdash}, m), \cdots, (\mathfrak{l}_{1\dashv}, m'')^{\mathbb{N}}\rangle$, and the terminating executions of $\mathbb{P}(C_2)$ are of the form $\langle(\mathfrak{l}_{2\vdash}, m''), \cdots, (\mathfrak{l}_{2\dashv}, m')^{\mathbb{N}}\rangle$. Then, from the induction hypothesis, $(m, m'') \in T_{\mathbb{M}}(C_1)$ and $(m'', m') \in T_{\mathbb{M}}(C_2)$; and from (2.5), $(m, m') \in T_{\mathbb{M}}(C)$.

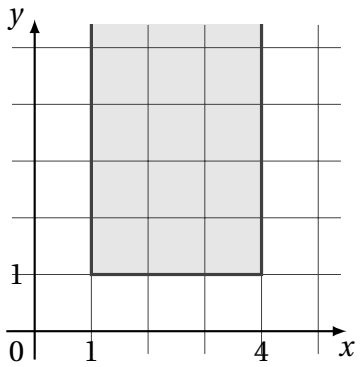
(iii) For a branching $C = C_1 + C_2$ (Fig. 2.8c), a terminating execution $\tau \in [\![\mathbb{P}(C)]\!]$ iff $\tau = \langle(\mathfrak{l}_{\vdash}, m), \tau', (\mathfrak{l}_{\dashv}, m')^{\mathbb{N}}\rangle$ where $m, m' \in \mathbb{M}$, $\tau' = \langle(\mathfrak{l}'_{\vdash}, m), \cdots, (\mathfrak{l}'_{\dashv}, m')\rangle \in \mathbb{S}^*$, and also $\langle\tau', (\mathfrak{l}'_{\dashv}, m')^{\mathbb{N}}\rangle \in [\![\mathbb{P}(C_1)]\!] \cup [\![\mathbb{P}(C_2)]\!]$. From the induction hypothesis, this can be iff $(m, m') \in T_{\mathbb{M}}(C_1) \cup T_{\mathbb{M}}(C_2)$; From (2.5), this can be iff $(m, m') \in T_{\mathbb{M}}(C)$.

(iv) For a loop $C = C'^*$ (Fig. 2.8d), a terminating execution $\tau \in [\![\mathbb{P}(C)]\!]$ iff

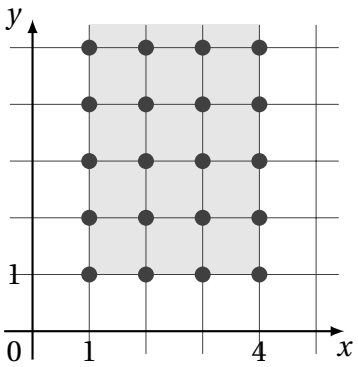
$$\tau = \langle(\mathfrak{l}_{\vdash}, m), (\mathfrak{l}_{\dashv}, m)\rangle, \text{ or}$$
$$\tau = \langle(\mathfrak{l}_{\vdash}, m_0), \tau'_0, \tau'_1, \cdots, \tau'_{k-1}, (\mathfrak{l}_{\dashv}, m_k)\rangle, \text{ where } k \geq 1 \text{ and for } i = 0..k-1$$
$$\tau'_i \in \mathbb{S}^*, \tau'_i = \langle(\mathfrak{l}'_{\vdash}, m_i) \cdots (\mathfrak{l}'_{\dashv}, m_{i+1}), (\mathfrak{l}_{\vdash}, m_{i+1})\rangle$$

The latter can be iff for every $i = 0..k-1$, $\langle\tau'_i, (\mathfrak{l}'_{\dashv}, m_{i+1})^{\mathbb{N}}\rangle \in [\![\mathbb{P}(C')]\!]$, i.e., iff (from the induction hypothesis) $(m_i, m_{i+1}) \in T_{\mathbb{M}}(C')$, i.e., iff $(m_0, m_k) \in T_{\mathbb{M}}(C')^k$.

That is, $\tau \in [\![\mathbb{P}(C)]\!]$ iff $\tau_{(0)} = (\mathfrak{l}_{\vdash}, m)$, $\tau_{(\dashv)} = (\mathfrak{l}_{\vdash}, m')$, and $(m, m') \in T_{\mathbb{M}}(C')^j$ for some $j \geq 0$. This is equivalent to saying that $(m, m') \in \bigcup_{j\geq 0} T_{\mathbb{M}}(C')^j = \mathrm{lfp}_{\subseteq} \lambda X.\Delta_{\mathbb{M}} \cup (X \circ T_{\mathbb{M}}(C')) = T_{\mathbb{M}}(C)$. □



(a) As a set of rational points.



(b) As a set of integer points.

Figure 2.9: An element of the interval domain.

2.B Memory Abstract Domains

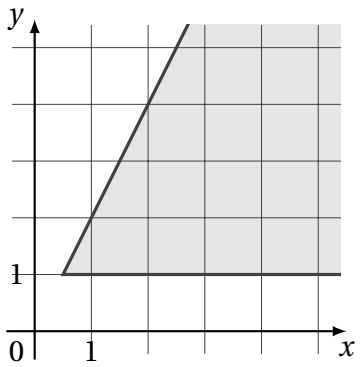
In this section, we briefly discuss some important abstract domains, i.e., ways to efficiently represent sets of memory states of different kinds of programs.

Numeric Domains

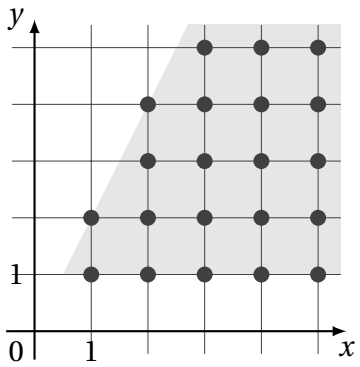
For a numeric program, a concrete (non-error) memory state is a map from the program variables $\mathbb{V}$ to their values which usually come from a subset of integer or rational numbers ($\mathbb{Z}$ or $\mathbb{Q}$). One can see a concrete state as a point in the n-dimensional space, where $n = |\mathbb{V}|$ (the number of programs variables; it is assumed to be finite). Then, numeric abstract domains usually correspond to different ways of finitely representing convex sets of points.

Interval Domain is probably the simplest one. An element of the interval domain maps every program variable to a (not necessarily bounded) range of possible values. For example, in a program with two variables, x and y, the object $\langle x : [1;4], y : [1;+\infty)\rangle$ is an element of the interval domain representing the set of memory states $\{m \mid 1 \leq m(x) \leq 4 \wedge m(y) \geq 1\}$. Fig. 2.9 displays this element as a 2-dimensional plot.

Polyhedral Domain represents an abstract memory state as a conjunction of linear inequalities (strict or non-strict) over program variables. An example of an element in the polyhedral domain is the expression $(y \leq 2x \wedge y \geq 1)$. This element is shown in Fig. 2.10. Implementations of the domain internally use *dual representation* of polyhedra, storing both the set of constraints (as a matrix of equation coefficients) and the



(a) As a set of rational points.



(b) As a set of integer points.

Figure 2.10: An element of polyhedral domain.

set of so called the *generators*: vertices (extreme points) and directions of unbounded edges (extreme rays). In our sample polyhedron, $(0.5, 1)$ is an extreme point, and the extreme rays can be the vectors $(1, 0)$ and $(1, 2)$.

Dual representation is used since different operations can be implemented more optimally using different representations. For example, (the constraints of) the intersection of two polyhedra can be produced by taking the union of their sets of constraints (and then removing redundant ones); the (generators of the) convex hull of two polyhedra can be produced by taking the union of their generators; it also is easier to minimize a system of constrains when generators are known; etc.

One of the first applications of polyhedra to program analysis was described by Patrick Cousot and Nicolas Halbwachs [CH78], but the underlying representations and algorithms were previously known in the field of linear optimization (e.g., a widely used algorithm to convert between the set of constraints and the set of generators is attributed to N. V. Chernikova [Che64; Che65; Che68]).

An interesting and useful feature of the polyhedral domain is that the convex hull operation (which over-approximates the union of the points two polyhedra and becomes join operation for the domain) can invent new linear relations. For example, observe the Fig. 2.11. In the figure, two rectangular polyhedra: $(1 \leq x \leq 3 \wedge 1 \leq y \leq 3)$ and $(4 \leq x \leq 6 \wedge 3 \leq y \leq 5)$ are shown with darker gray background. Their convex hull is shown with lighter gray background. Observe that it is not a rectangle any more, but also bounds the difference of $2x$ and $3y$. This allows an analysis to materialize linear relations that are not explicitly present in the program. For example, for the program in Fig. 2.12, a polyhedral analysis will be able to assert that after the loop finishes, $x = y$.

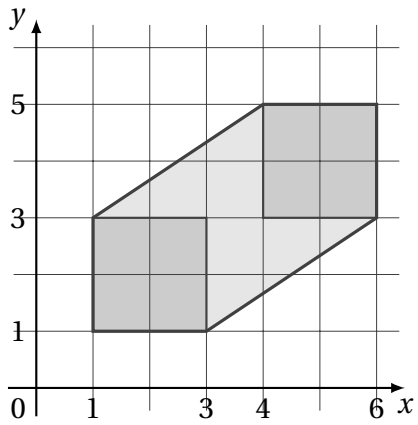


Figure 2.11: The convex hull of two polyhedra.

```
1 | x ← y ← 0;
2 | while(?) {
3 |     x ← x + 1;
4 |     y ← y + 1;
5 | }
```

Figure 2.12: A program that simultaneously updates two variables.

The interval and polyhedral domains, in a way, are the two extremes. Polyhedra can store arbitrary linear relations between variables, which benefits the precision of an analysis, but makes it computationally expensive, especially when the number of dimensions (program variables) is high (e.g., a bounded rectangle in n dimensions already contains 2^n points). Interval computations are inexpensive, but lose all the information about relations between variables and produce imprecise analyses. A number of trade-offs between efficiency and precision have been proposed. For example, the octagon [Min06] domain allows constraints of the form $\pm x \pm y \leq c$ that relate two variables x and y, and a constant c.

Domain of Linear Congruences allows to assert divisibility of a linear expression by a constant. Often, it is used in conjunction with polyhedra (forming a so called reduced product of the two domains): an abstract memory state is represented by a pair consisting of a polyhedron and a set of linear congruences. An example of an element of the product domain is shown in Fig. 2.13 and represents the constraints $\big(x \geq 0 \wedge y \geq 0 \wedge (x = 0 \mod 2) \wedge (y = 0 \mod 2)\big)$.

Shape Analysis with 3-valued Logic

Here, we give a brief introduction on how analyses based on 3-valued logic represent heaps of programs. For more information on shape analysis with 3-valued logic, please refer to Sagiv et al. [SRW02] and related papers [RSL10; Arn+06; LMS04].

This framework is designed to represent heaps that contain linked data structures:

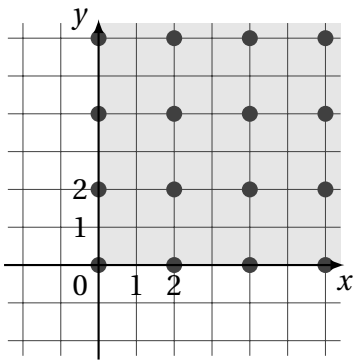


Figure 2.13: An element of the product of polyhedra and linear congruences.

sets of heap cells (where a *cell* is a continuous region of dynamic memory, e.g., an instance of Java class or a heap-allocated instance of C structure) that are pointed to by program variables and also store pointers to each other. This allows to analyse programs that manipulate singly- or doubly-linked lists, trees, etc.

In the framework, abstract heaps are represented with so called *3-valued structures*, i.e., models of 3-valued first-order logic with transitive closure. Every individual represents either a single heap cell or a set of heap cells that share some properties. Pointer variables are represented by unary predicates: the predicate is true for the cell where the variable points. Pointer fields are represented by binary predicates: the predicate is true for those pairs of cells where the corresponding field of one cell points to another cell. The analysis also maintains in the form of predicates additional information about the heap: whether the cells are reachable from each other or from some pointer variable, whether cells lie on a cycle, whether some condition is true of the cells, and so on[19]. The choice of these additional predicates depends on the kind of data structures that a program manipulates. For example, if we constrain the input of a program only to acyclic singly-linked lists, we will use a different set of predicates than if we allowed cyclic lists or doubly-linked lists. The authors of TVLA offer pre-made sets of predicates for different data structures, and we will not discuss those in detail.

Three-valued structures can be displayed as *shape graphs*, and an example is shown in Fig. 2.14. The graph represents an acyclic singly-linked list with two or more elements and can be interpreted as follows. The left node represents a single cell which is the head of the list and is pointed to by pointer variables x and y. The text $c = {}^1/_2$ means

[19] Soundly maintaining the strongest possible information during the analysis is an important and complicated task.

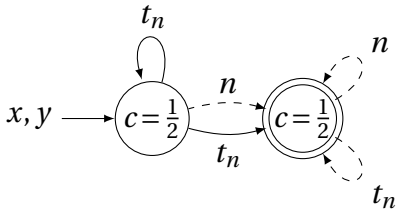


Figure 2.14: Acyclic list with 2+ elements.

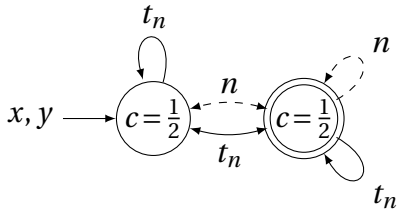


Figure 2.15: Cyclic list with 2+ elements.

that some condition c might or might not be true for the head – we do not know. The right node is displayed with double border and represents a finite non-empty set of cells that constitute the tail of the list (so called summary node). The dotted edge annotated by n between the head and the tail means that the pointer field n of the head points to some node of the tail, but not to all of them. The analysis is usually instructed that predicate n induces a function, but this is usually not reflected in the shape graph. In our case, the analysis also keeps track of reachability between cells with the predicate t_n. Solid t_n-edge between the head and the tail means that all cells of the tail are reachable from the head by traversing the n-pointers. Dotted n- and t_n-loops on the tail mean that there are pointers and reachability between some pairs of cells in the tail but not between all of them. Absence of n- and t_n edges from the tail to the head means that no cell in the tail points to or can reach head. In this case, the analysis is also instructed that there are no *shared* cells, i.e., every cell is pointed to by at most one cell. The above is sufficient for Fig. 2.14 to represent exactly the set of acyclic singly-linked lists with two or more elements. Similarly, Fig. 2.15 represents a set of cyclic lists with two or more elements. Predicates like n are called *core* predicates, and they are the main carriers of the information about the structure. Predicates like t_n are called *instrumentation* predicates. They have definitions in terms of other predicates, and in a concrete structure (without dashed edges or summary nodes), they are redundant and provide no additional information. For an abstract structure, instrumentation predicated do provide additional information and do restrict the set of possible concretizations. Abstract transformers are designed in such a way as to preserve as much information as possible and to update the instrumentation predicates in the most precise way.

Separation Logic

Finally, we mention separation logic [Rey02] – a successful way of representing abstract heaps of programs. The central connective in this logic is *separating conjunction*. For a pair of statements, it asserts that there is a partition of the heap (into two disjoint parts), s.t. the first statement holds for one part, and the second statement holds for another part. We do not use separation logic in this work though (we use 3-valued logic to analyse heap-manipulating programs).

Chapter 3

Finding Existential Recurrent Sets with Backward Analysis

In this chapter we present an algorithm that allows to under-approximate existential recurrent sets of individual loops in unstructured programs.

The fixed point characterization of existential recurrent set (given in Lemma 2.5) is actually hard to apply in practice in an under-approximating way. The main issue is that backward analysis with the predecessors transformer (pre) introduces disjunctions that are hard to under-approximate in most domains. Let us look at the characterization again:

$$R = \mathrm{gfp}_{\subseteq}\, \lambda X.\big(\{s \in \mathbb{S} \mid s \text{ is non-error and not final}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X)\big)$$

In this chapter, we will be approximating recurrent sets of individual loops. In particular, in a loop, every location is not final (because every location has a successor in the loop) and thus (if now $\mathbb{P}$ denotes a single loop)

$$R = \mathrm{gfp}_{\subseteq}\, \lambda X.\{s \in \mathbb{S} \mid s \text{ is non-error}\} \cap \mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X)$$

Since the set predecessors of an error state includes error states, but the predecessors of non-error states are non-error

$$= \mathrm{gfp}_{\subseteq}\, \lambda X.\mathrm{pre}(T_{\mathbb{S}}(\mathbb{P}), X \setminus (\mathbb{L} \times \{\varepsilon\}))$$

One can think of this as a greatest fixed point in the domain of non-error states $\mathscr{P}(\mathbb{L} \times (\mathbb{M} \setminus \{\varepsilon\}))$. The computation then creates a chain of approximants of R. In this chapter,

we focus on numeric programs and use polyhedra (conjunctions of linear inequalities) to represent memory states. That is, one can decide that every approximant of R should be an element of the domain $\mathbb{L} \to \mathbb{D}_{\text{poly}}$ and maps program locations to polyhedra (note that since we characterised R in the domain of non-error states, our abstract domain does not have to be able to represent error states). Let us denote the current value of this map in a computation (the current approximant of the fixed point) by d. Initially, d maps every location to a polyhedron without constraints ($\top$):

$$d(l) = \top, \text{ for } l \in \mathbb{L}$$

In practice, the computation proceeds using chaotic iteration [Bou93b]. That is, in a computation step, instead of updating all entries of the map at once as in

$$d \leftarrow d \sqcap \text{pre}(\mathbb{P}, d), \text{ until } d \text{ stabilizes}$$

we pick program locations one by one in some order and sequentially update their entries in d (in Appendix 3.B, we make a note on the use of chaotic iteration). Thus, in every step, we pick a location l and perform the update:

$$d(l) \leftarrow d(l) \sqcap \bigsqcup_{l' \in \text{succ}(l)} \text{pre}(\mathfrak{c}(l, l'), d(l'))$$

Note that here, we join the pre-conditions (w.r.t. corresponding edges) of all successors of l in the program graph. The resulting set is a disjunction of convex sets[1] and thus is not convex in general and might not be represented exactly by a single polyhedron. This would not be a problem in an over-approximating analysis where we can just take the convex hull of all disjuncts. But in an under-approximating analysis, we would need to come up with a heuristic that would produce some good (in a sense that it would allow to produce some non-empty existential recurrent set) convex under-approximation of a disjunction of convex elements, and this is not easy to do.

It is actually the case that for many programs, recurrent sets when projected to individual program locations are not convex and, cannot be reasonably approximated by

[1] In program analysis, the term *convex* can be used in a loose sense, meaning *expressible as a conjunction of some atomic facts*. When we take atomic facts to be linear inequalities, sets that are convex in this sense (i.e., polyhedra) are also convex in the geometrical sense.

```
1 while (x ≠ 0) {
2   if (x < 0) {
3     x ← x − 1;
4     x ← −x;
5   } else {
6     x ← x + 1;
7     x ← −x;
8   }
9 }
```

Figure 3.1: A program where a non-terminating execution alternates between regions $x > 0$ and $x < 0$. Thus a recurrent set is not convex.

a map from program locations to single polyhedra. Consider a program that is shown in pseudocode in Fig. 3.1. For this program (and for all other examples of numeric programs in this work), let us assume that the variable x takes integer values. While the value of the variable x is not 0, the program will increase x, if it is positive, or decrease it, if it is negative – and then invert its value. Thus, if x starts with a nonzero value, the loop will run forever, infinitely increasing the absolute value of x. At the same time, at the location corresponding to the head of the loop, the value of x alternates between the regions where $x > 0$ and $x < 0$, thus there is no convex recurrent set.

In this chapter, we propose an analysis that attempts to work around this issue. The analysis will maintain multiple convex elements per program location, thus allowing to represent non-convex recurrent sets. Every element will correspond to a different set of paths through the program represented as an element of a finite *path domain* (which sets the limit to a number of disjuncts that we keep per program location). This technique is called *trace partitioning* (for trace partitioning in forward analysis, see [MR05]). Also, we will allow the analysis to perform some over-approximating operations (e.g., to sometimes join convex polyhedra by taking a convex hull). In particular, we will allow to join memory states corresponding to the same set of paths. This way, using not necessarily under-approximate backward analysis, we will infer a (potentially unsound) *candidate existential recurrent set*. Then, we will check the candidate for soundness and possibly refine it using an over-approximate forward analysis.

For this chapter, we will make a number of assumptions on the memory domain.

In particular, we will assume that there exists a meet operation that allows backward analysis to build a descending chain; then, we will use lower widening to ensure convergence of backward analysis. This is suitable for numeric domains, but non-numeric domains may employ different techniques. For example, in shape analysis with 3-valued logic [SRW02], convergence is due to the use of a finite domain of *bounded structures*. Our backward analysis would need to be modified to be applicable to this and similar domains.

We report experimental results in Chapter 5

3.1 Abstract Domain of the Analysis

Let the *memory abstract domain* of the analysis be $\mathbb{D}_\mathfrak{m}$, with least element $\bot_\mathfrak{m}$, greatest element $\top_\mathfrak{m}$, partial order $\sqsubseteq_\mathfrak{m}$, and join $\sqcup_\mathfrak{m}$. Every *element*, or abstract memory state, $a \in \mathbb{D}_\mathfrak{m}$ represents a set of memory states $\gamma_\mathfrak{m}(a) \subseteq \mathbb{M}$. We lift *concretization* to sets of abstract memory states: for $A \subseteq \mathbb{D}_\mathfrak{m}$, $\gamma_\mathfrak{m}(A) = \bigcup\{\gamma_\mathfrak{m}(a) \mid a \in A\}$. In the context of this chapter, $\mathbb{D}_\mathfrak{m}$ can be assumed to be a polyhedral domain where an element is a conjunction of linear inequalities over the program variables (or some other numeric abstract domain).

We assume that we are given the over-approximate versions of the transformers post, pre, and eval, s.t. for an atomic statement $C \in \mathbb{A}$, an element $a \in \mathbb{D}_\mathfrak{m}$, and a memory-state formula θ,

$$\gamma_\mathfrak{m}(\text{post}^\mathfrak{m}(C, a)) \supseteq \text{post}(T_\mathbb{M}(C), \gamma_\mathfrak{m}(a))$$
$$\gamma_\mathfrak{m}(\text{pre}^\mathfrak{m}(C, a)) \supseteq \text{pre}(T_\mathbb{M}(C), \gamma_\mathfrak{m}(a))$$
$$\text{eval}^\mathfrak{m}(\theta, a) \sqsupseteq_\mathscr{K} \text{eval}(\theta, \gamma(a))$$

In this chapter, we are mostly interested in numeric programs, which, apart from passive and assumption statements, can use:

(i) a *deterministic assignment* $x \leftarrow expr$, which assigns the value of an expression *expr* to a program *variable* x;

(ii) a *nondeterministic assignment*, or *forget operation*, $x \leftarrow *$, which assigns a nondeterministically selected value to a program variable x.

For $\text{eval}^{\mathfrak{m}}$, we should note that normally, it is given for atomic formulas, and for arbitrary formulas it is defined by induction over the formula structure, using 3-valued logical operators, possibly over-approximate w.r.t. $\sqsubseteq_{\mathscr{K}}$. For example, let a be the conjunction of linear inequalities: $(-1 \leq x \leq 1) \wedge (-1 \leq y \leq 1)$, which can be seen as a polyhedron in two dimensions. Let the formula $\theta = (x > 0) \vee (y > 1)$. In order to compute $\text{eval}^{\mathfrak{m}}(\theta, a)$, we will normally do the following:

$$\text{eval}^{\mathfrak{m}}(\theta, a) = \text{eval}^{\mathfrak{m}}(x > 0, a) \vee \text{eval}^{\mathfrak{m}}(y > 1, a) = {}^{1}/_{2} \vee 0 = {}^{1}/_{2}$$

In this chapter, we also make the following assumptions on $\mathbb{D}_{\mathfrak{m}}$. We assume there exists a meet operation, s.t. for $a_1, a_2 \in \mathbb{D}_{\mathfrak{m}}$, $a_1 \sqcap_{\mathfrak{m}} a_2 \sqsubseteq_{\mathfrak{m}} a_1$ and $a_1 \sqcap_{\mathfrak{m}} a_2 \sqsubseteq_{\mathfrak{m}} a_2$. This allows producing descending chains in $\mathbb{D}_{\mathfrak{m}}$ and performing approximation of greatest fixed points even with non-monotonic abstract transformers[2]. If $\mathbb{D}_{\mathfrak{m}}$ admits infinite descending chains, we assume there exists *lower widening* operation $\underline{\triangledown}_{\mathfrak{m}}$. Similarly, if $\mathbb{D}_{\mathfrak{m}}$ admits infinite ascending chains, we assume there exists *widening* operation $\triangledown_{\mathfrak{m}}$. Continuing the discussion started in Section 2.3.3, in this chapter, we require $\mathbb{D}_{\mathfrak{m}}$ to be able to represent erroneous states. Although the main step of the analysis uses backward analysis and involves only non-error abstract states, the subsequent refinement step needs to be able to detect abstract states with erroneous successors.

To produce a standard over-approximate analysis one would then move to the domain $\mathbb{L} \to \mathbb{D}_{\mathfrak{m}}$, where every element represents a set of program states partitioned with locations. For the purpose of trace partitioning, we take an additional step to introduce what we call a *path abstract domain* $\mathbb{D}_{\mathfrak{p}}$, with least element $\bot_{\mathfrak{p}}$, greatest element $\top_{\mathfrak{p}}$, partial order $\sqsubseteq_{\mathfrak{p}}$, join $\sqcup_{\mathfrak{p}}$ and meet $\sqcap_{\mathfrak{p}}$. Every *element*, or abstract path, $q \in \mathbb{D}_{\mathfrak{p}}$ represents a set of paths $\gamma_{\mathfrak{p}}(q) \subseteq \Pi$. We introduce over-approximate versions of post and

[2]Although, in Chapter 2 we, for simplicity, preferred to assume the domains to be complete lattices, not all useful abstract domains actually are. For example, if in the context of shape analysis with 3-valued logic we consider the domain of (sets of) *bounded* structures, we will find that the greatest lower bound does not always exist. More specifically, the greatest lower bound will always exist in the domain of (sets of) all 3-valued structures, but for two (sets of) bounded structures, their greatest lower bound is not necessary bounded. In this case, we could try to define *some* lower bound operation (we can always make it produce $\bot_{\mathfrak{m}}$ for some pairs of arguments), but we think it is better not to do so. The abstract meet operation is supposed to be an approximation of concrete set intersection, and we believe that it may be better to have a meaningful over-approximation of set intersection (e.g., in this case, taking a meet of 3-valued structures and applying canonical abstraction) than to have an artificially introduced under-approximate meet. This means that for such domains we might not have a way to produce descending chains, and the analysis of this chapter cannot be applied in such domains directly (but we anticipate that it can be adapted).

pre, s.t. for an edge $e \in \mathbb{E}$ and an element $q \in \mathbb{D}_\mathfrak{p}$,

$$\gamma_\mathfrak{p}(\text{post}^\mathfrak{p}(e, q)) \supseteq \text{post}(T_\Pi(\mathbb{P})|_e, \gamma_\mathfrak{p}(q))$$
$$\gamma_\mathfrak{p}(\text{pre}^\mathfrak{p}(e, q)) \supseteq \text{pre}(T_\Pi(\mathbb{P})|_e, \gamma_\mathfrak{p}(q))$$

where

$$T_\Pi(\mathbb{P})|_e = \{((p, i), (p, i+1)) \in \Pi \times \Pi \mid (p_{(i)}, p_{(i+1)}) = e\}$$

that is, this is a restriction of the transition relation on paths to an edge $e \in \mathbb{E}$. For our purposes, we also assume that $\mathbb{D}_\mathfrak{p}$ is finite, and there exists abstraction function $\alpha_\mathfrak{p}$ that, together with $\gamma_\mathfrak{p}$ forms a Galois connection between $\mathbb{D}_\mathfrak{p}$ and $\mathscr{P}(\Pi)$. This allows to partition memory states with elements of $\mathbb{L} \times \mathbb{D}_\mathfrak{p}$, similarly to how a standard analysis partitions memory states with locations.

Constructing the Abstract Domain of the Analysis.

Given a memory abstract domain $\mathbb{D}_\mathfrak{m}$ and a path abstract domain $\mathbb{D}_\mathfrak{p}$ with required properties, let us first construct an auxiliary abstract domain $\mathbb{D}_{\mathfrak{mp}} \subseteq \mathbb{D}_\mathfrak{p} \rightharpoonup \mathbb{D}_\mathfrak{m}$ (where $\rightharpoonup$ denotes a partial function). We require that every element $D \in \mathbb{D}_{\mathfrak{mp}}$ is what we call *reduced*: for every $q \in \text{dom}(D)$, $q \neq \bot_\mathfrak{p}$ and $D(q) \neq \bot_\mathfrak{m}$; and for every pair of abstract paths $q_1, q_2 \in \text{dom}(D)$, $q_1 \sqcap_\mathfrak{p} q_2 = \bot_\mathfrak{p}$. Intuitively, D is a collection of abstract memory states partitioned with *disjoint* abstract paths. For every partial function $D' : \mathbb{D}_\mathfrak{p} \rightharpoonup \mathbb{D}_\mathfrak{m}$, we can produce a reduced element $D = \text{reduce}(D') \in \mathbb{D}_{\mathfrak{mp}}$. To do so, we remove bottom elements and then repeatedly join the pairs from D' (thinking of a function as of a set of pairs) that have non-disjoint abstract paths. This procedure is shown in Fig. 3.2.

The top element $\top_{\mathfrak{mp}} = \{\top_\mathfrak{p} \mapsto \top_\mathfrak{m}\}$; the bottom element $\bot_{\mathfrak{mp}}$ is the empty partial function. The partial order $\sqsubseteq_{\mathfrak{mp}}$ is point-wise. For $D_1, D_2 \in \mathbb{D}_{\mathfrak{mp}}$,

$$D_1 \sqsubseteq_{\mathfrak{mp}} D_2 \text{ iff } \forall (q_1, a_1) \in D_1.\ \exists (q_2, a_2) \in D_2.\ q_1 \sqsubseteq_\mathfrak{p} q_2 \wedge a_1 \sqsubseteq_\mathfrak{m} a_2$$

Join is just a set union. For $D_1, D_2 \in \mathbb{D}_{\mathfrak{mp}}$,

$$D_1 \sqcup_{\mathfrak{mp}} D_2 = \text{reduce}(D_1 \cup D_2)$$

When taking meet of $D_1, D_2 \in \mathbb{D}_{\mathfrak{mp}}$, we meet the tuples from D_1 and D_2 pair-wise. For

Algorithm: Reduce
Input: Non-reduced element $D' : \mathbb{D}_{\mathfrak{p}} \rightharpoonup \mathbb{D}_{\mathfrak{m}}$
Output: Reduced element $D \in \mathbb{D}_{\mathfrak{mp}}$

1 $D \leftarrow D'$
2 **for** $d \in D$, s.t. $d = (\bot_{\mathfrak{p}}, a) \vee d = (q, \bot_{\mathfrak{m}})$ **do**
3 $\quad D \leftarrow D \setminus \{d\}$
4 **endfor**
5 **for** $(q_1, a_1), (q_2, a_2) \in D$, s.t. $q_1 \sqcap_{\mathfrak{p}} q_2 \neq \bot_{\mathfrak{p}}$ **do**
6 $\quad D \leftarrow (D \setminus \{(q_1, a_1), (q_2, a_2)\}) \cup (q_1 \sqcup_{\mathfrak{p}} q_2, a_1 \sqcup_{\sharp} a_2)$
7 **endfor**

Figure 3.2: Reducing a partial function to an element of $\mathbb{D}_{\mathfrak{mp}}$.

$D_1, D_2 \in \mathbb{D}_{\mathfrak{mp}}$,

$$D_1 \sqcap_{\mathfrak{mp}} D_2 = \{(q_1 \sqcap_{\mathfrak{p}} q_2, a_1 \sqcap_{\sharp} a_2) \mid (q_1, a_1) \in D_1 \wedge (q_2, a_2) \in D_2 \wedge q_1 \sqcap_{\mathfrak{p}} q_2 \neq \bot_{\mathfrak{p}} \wedge a_1 \sqcap_{\mathfrak{m}} a_2 \neq \bot_{\mathfrak{m}}\}$$

As both D_1 and D_2 are reduced (all the elements of $\mathrm{dom}(D_1)$ are incomparable and so are the elements of $\mathrm{dom}(D_2)$), it follows that $D_1 \sqcap_{\mathfrak{mp}} D_2$ is reduced, as the all elements of the form $q_1 \sqcap_{\mathfrak{p}} q_2$ where $q_1 \in \mathrm{dom}(D_1)$ and $q_2 \in \mathrm{dom}(D_2)$ are either bottom or incomparable.

Abstract post-condition and predecessor operations are path-wise. For $e \in \mathbb{E}$ and $D \in \mathbb{D}_{\mathfrak{mp}}$,

$$\mathrm{post}^{\mathfrak{mp}}(e, D) = \mathrm{reduce}(\{(\mathrm{post}^{\mathfrak{p}}(e, q), \mathrm{post}^{\mathfrak{m}}(\mathfrak{c}(e), a) \mid (q, a) \in D\})$$
$$\mathrm{pre}^{\mathfrak{mp}}(e, D) = \mathrm{reduce}(\{(\mathrm{pre}^{\mathfrak{p}}(e, q), \mathrm{pre}^{\mathfrak{m}}(\mathfrak{c}(e), a) \mid (q, a) \in D\})$$

Widening and lower widening are path-wise. For $D_1, D_2 \in \mathbb{D}_{\mathfrak{mp}}$, s.t. $D_1 \sqsubseteq_{\mathfrak{mp}} D_2$,

$$D_1 \triangledown_{\mathfrak{mp}} D_2 = \{(q, a') \mid (q, a) \in D_2 \text{ and if } q \notin \mathrm{dom}(D_1) \text{ then } a' = a \text{ else } a' = D_1(q) \triangledown_{\mathfrak{m}} D_2(q)\}$$
$$D_2 \underline{\triangledown}_{\mathfrak{mp}} D_1 = \{(q, a') \mid (q, a) \in D_1 \text{ and if } q \notin \mathrm{dom}(D_2) \text{ then } a' = a \text{ else } a' = D_2(q) \underline{\triangledown}_{\mathfrak{m}} D_1(q)\}$$

In practice, when widening is only applied after a certain delay, it may make sense to

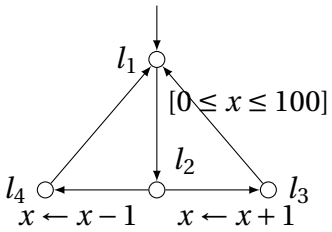


Figure 3.3: Loop containing single non-deterministic branching statement.

define a more aggressive lower widening.

$$D_2 \underline{\triangledown}_{\mathfrak{mp}} D_1 = \{(q, a') \mid q \in \mathrm{dom}(D_1) \wedge q \in \mathrm{dom}(D_2) \wedge a' = D_2(q) \underline{\triangledown}_{\mathfrak{m}} D_1(q)\}$$

We observed that with a sufficient widening delay, the abstract paths that are present in D_1 and not present in D_2 are unlikely to be interesting, and removing them actually helps the algorithm.

Then, the abstract domain of our analysis (in this domain, backward analysis will be performed) is $\mathbb{D}_\sharp = \mathbb{L} \to \mathbb{D}_{\mathfrak{mp}}$. That is, one can think of an element $D \in \mathbb{D}_\sharp$ as a collection of abstract program states partitioned by location and abstract path. In a sense, an element $D \in \mathbb{D}_\sharp$ answers the question, *"What should be the memory state at a given location, assuming that from this state the program may take only certain paths"*. The construction that produces the domain $\mathbb{L} \to \mathbb{D}_{\mathfrak{mp}}$ from $\mathbb{D}_{\mathfrak{mp}}$ is standard in abstract interpretation (usually, it is applied to produce $\mathbb{L} \to \mathbb{D}_{\mathfrak{m}}$ from $\mathbb{D}_{\mathfrak{m}}$) and we move its description to Appendix 3.A. All the operations in $\mathbb{D}_\sharp$ are just location-wise applications of the corresponding operations in $\mathbb{D}_{\mathfrak{mp}}$. We note though in $\mathbb{D}_\sharp$, the post-condition $\mathrm{post}^\sharp(\mathbb{P}, \cdot)$ and predecessor operation $\mathrm{pre}^\sharp(\mathbb{P}, \cdot)$ are taken with respect to the whole program.

3.2 Path Domain

For the path domain, in this work, we use finite sequences of *future branching choices*. A *branching point* is a location $l \in \mathbb{L}$, s.t. there exists at least two edges from l. A *branching choice* is an edge $(l, l') \in \mathbb{E}$, s.t. l is a branching point. For example, consider a program fragment in Fig. 3.3. The location l_2 is a branching point, and the edges (l_2, l_3) and (l_2, l_4) are branching choices.

We denote the set of all branching choices by $\mathbb{E}_{\mathfrak{b}} \subseteq \mathbb{E}$. For every non-bottom element $q \in \mathbb{D}_{\mathfrak{p}}$, q is a finite sequence of branching choices: $q = \langle e_0, e_1, \ldots, e_n \rangle \in \mathbb{E}_{\mathfrak{b}}^*$; top element

$\top_{\mathfrak{p}}$ is the empty sequence $\langle\rangle$; and bottom is a distinguished element $\bot_{\mathfrak{p}} \notin \mathbb{E}_{\mathfrak{b}}^*$. That is, $\mathbb{D}_{\mathfrak{p}} \subseteq \mathbb{E}_{\mathfrak{b}}^* \cup \{\bot_{\mathfrak{p}}\}$.

For $q_1, q_2 \in \mathbb{D}_{\mathfrak{p}}$,

$$q_1 \sqsubseteq_{\mathfrak{p}} q_2 \text{ iff } q_1 = \bot_{\mathfrak{p}} \text{ or } q_2 \text{ is a prefix of } q_1$$

For $q_1, q_2 \in \mathbb{D}_{\mathfrak{p}}$, join is

$$q_1 \sqcup_{\mathfrak{p}} q_2 = \begin{cases} q_2, \text{ if } q_1 = \bot_{\mathfrak{p}} \\ q_1, \text{ if } q_2 = \bot_{\mathfrak{p}} \\ \text{the longest common prefix of } q_1 \text{ and } q_2, \text{ otherwise} \end{cases}$$

For $q_1, q_2 \in \mathbb{D}_{\mathfrak{p}}$, meet is

$$q_1 \sqcap_{\mathfrak{p}} q_2 = \begin{cases} q_1, \text{ if } q_1 \sqsubseteq_{\mathfrak{p}} q_2 \\ q_2, \text{ if } q_2 \sqsubseteq_{\mathfrak{p}} q_1 \\ \bot_{\mathfrak{p}}, \text{ otherwise} \end{cases}$$

For example, let $q_1 = \langle (l_2, l_3), (l_2, l_4) \rangle$. It represents the set of paths where the first two times control reaches l_2, it is transferred to l_3, and after that the path is not constrained. Also, let $q_2 = \langle (l_2, l_3), (l_2, l_4) \rangle$. In a similar way, represents the set of paths, where the first time control reaches l_2, it is transferred to l_3, the second time – it is transferred to l_4, and after that the path is not constrained. Informally, can see that q_1 and q_2 represent disjoint sets of paths. Formally, neither is the prefix of the other one. Thus, $q_1 \not\sqsubseteq_{\mathfrak{p}} q_2$, $q_2 \not\sqsubseteq_{\mathfrak{p}} q_1$, and $q_1 \sqcap_{\mathfrak{p}} q_2 = \bot_{\mathfrak{p}}$. At the same time, q_1 and q_2 have the common prefix $\langle (l_2, l_3) \rangle$. Thus, the join of these two abstract paths is $q_1 \sqcup_{\mathfrak{p}} q_2 = \langle (l_2, l_3) \rangle$. This represents the set of paths, where the first time control reaches l_2, it is transferred to l_3, and after that the path is not constrained. Informally, one can see that this contains the sets, represented by q_1 and q_2.

We assume that $\mathbb{D}_{\mathfrak{p}}$ only contains a finite number of elements. The following construction worked reasonably well in our experiments. We assume that every element $q \in \mathbb{D}_{\mathfrak{p}}$ is *bounded*, in a sense that every branching choice $e \in \mathbb{E}_{\mathfrak{b}}$ appears in q at most k times, where $k \geq 1$ is a parameter of the domain. For a sequence of branching choices $q' \in \mathbb{E}_{\mathfrak{b}}^*$ (or $\in \mathbb{E}_{\mathfrak{b}}^{\mathbb{N}}$), we can produce a bounded element $\mathfrak{b}_k(q') \in \mathbb{D}_{\mathfrak{p}}$ by keeping the longest

bounded prefix of the sequence. For example, let $q = \langle (l_2, l_3), (l_2, l_4), (l_2, l_3), (l_2, l_4) \rangle$. Then, $\mathfrak{b}_1(q) = \langle (l_2, l_3), (l_2, l_4) \rangle$.

We can chose alternative definitions of a bounded element and the bounding function if required (as long as the domain of bounded elements is finite). For example, we could bound the total number of branching choices in an element, which can be useful to simplify the presentation of examples. We will use this definition in Example 3.1. With this definition, for $q = \langle (l_2, l_3), (l_2, l_4), (l_2, l_3), (l_2, l_4) \rangle$, we get $\mathfrak{b}_1(q) = \langle (l_2, l_3) \rangle$.

Intuitively, an element $q = \langle e_0, e_1, \ldots, e_n \rangle \in \mathbb{E}^*_{\mathfrak{b}}$ represents a set of paths where the next $n+1$ branching choices are e_0, e_1, etc, and after that the branching choices are not restricted. Formally, it represents the set of paths $\gamma_{\mathfrak{p}}(q) \subseteq \Pi$, s.t. $\pi = (\langle l_0, l_1, \ldots \rangle, i) \in \gamma_{\mathfrak{p}}(q)$ iff for $j = 0..n$, there exists a strictly increasing sequence of indices $\{x_j\}_{0 \le j \le n}$, s.t. $i \le x_0 < \ldots < x_n$ and every $\pi_{(x_j)}$ is a branching point, $(\pi_{(x_j)}, \pi_{(x_j+1)}) = e_j$, and for every index z, s.t. $i \le z < x_n$, if $z \notin \{x_j\}$, then $\pi_{(z)}$ is not a branching point.

The corresponding abstraction function can be defined as follows. For a single path $\pi = (\langle l_0, l_1, \ldots \rangle, i) \in \Pi$, intuitively, abstraction function extracts the next k branching choices. Formally, for $j \ge 0$, let $\{y_j\}_{j \ge 0}$ be a strictly increasing sequence of indices of branching points at or after position i, s.t. $i \le y_0 < y_1 < \ldots$, every $\pi_{(y_j)}$ is a branching point, and for every index $z \ge i$, if $z \notin \{y_j\}$, then $\pi_{(z)}$ is not a branching point. Then, the abstraction of π is $\alpha_{\mathfrak{p}}(\pi) = \mathfrak{b}_k(\langle (\pi_{(y_0)}, \pi_{((y_0)+1)}), (\pi_{(y_1)}, \pi_{((y_1)+1)}), \ldots \rangle)$. For a set of paths V, $\alpha_{\mathfrak{p}}(V) = \bigsqcup_{\mathfrak{p}} \{\alpha_{\mathfrak{p}}(\pi) \mid \pi \in V\}$.

For an edge $e \in \mathbb{E}$ and $q \in \mathbb{D}_{\mathfrak{p}}$, post-condition and predecessors are defined as follows (operator $\cdot$ denotes concatenation of sequences)

$$\mathrm{pre}^{\mathfrak{p}}(e, q) = \begin{cases} \bot_{\mathfrak{p}}, \text{ if } q = \bot_{\mathfrak{p}} \\ \mathfrak{b}_k(e \cdot q), \text{ if } q \neq \bot_{\mathfrak{p}} \text{ and } e \text{ is a branching choice} \\ q, \text{ otherwise} \end{cases}$$

$$\mathrm{post}^{\mathfrak{p}}(e, q) = \begin{cases} q', \text{ if } q = e \cdot q' \text{ for some } q' \in \mathbb{D}_{\mathfrak{p}} \\ \bot_{\mathfrak{p}}, \text{ if } q = e' \cdot q' \text{ for some } q' \in \mathbb{D}_{\mathfrak{p}} \text{ and } e' \neq e \\ \top_{\mathfrak{p}}, \text{ if } q = \top_{\mathfrak{p}} \\ \bot_{\mathfrak{p}}, \text{ if } q = \bot_{\mathfrak{p}} \end{cases}$$

Intuitively, $\text{pre}^{\mathfrak{p}}$ prepends a new branching choice to a path (and bounds the result), and $\text{post}^{\mathfrak{p}}$ removes a branching choice from a path, when possible. For example, for the fragment in Fig. 3.3, let $q = \langle (l_2, l_3), (l_2, l_4) \rangle$, and the bounding function is s.t. it keeps at most one occurrence of every branching choice. Then,

$$\begin{aligned}
&\text{pre}^{\mathfrak{p}}((l_1, l_2), q), && \text{as } (l_1, l_2) \text{ is not a branching choice} \\
&\text{pre}^{\mathfrak{p}}((l_2, l_3), q) = \mathfrak{b}_1(\langle (l_2, l_3), (l_2, l_3), (l_2, l_4) \rangle) = \langle (l_2, l_3) \rangle \\
&\text{post}^{\mathfrak{p}}((l_2, l_3), q) = \langle (l_2, l_4) \rangle \\
&\text{post}^{\mathfrak{p}}((l_2, l_4), q) = \bot_{\mathfrak{p}}, && \text{as } q \text{ does not start with } (l_2, l_4)
\end{aligned}$$

One can say that an abstract path $q \in \mathbb{D}_{\mathfrak{p}}$ predicts a bounded number of branching choices that an execution would make. We observe that our path domain works well for non-nested loops, and the bound k corresponds to the number of loop iterations, for which we keep the branching choices. In most our experiments, $k = 1$ or 2 was enough to find a recurrent set (for some programs, we had to use $k = 3$ or 4).

Note that the forward transformer $\text{post}^{\mathfrak{p}}$ leaves $\top_{\mathfrak{p}}$ unchanged. Thus, our backward analysis does use trace partitioning, but the forward pre-analysis does not (with the current path domain). The forward pre-analysis, is initialized with $f_0 = \{l_{\vdash} \mapsto \top_{\sharp}; l \neq l_{\vdash} \mapsto \bot_{\sharp}\}$, i.e., during the forward pre-analysis, every location is mapped either to $\bot_{\sharp}$ or to $\{\top_{\mathfrak{p}} \mapsto m\}$ for some $m \in \mathbb{D}_{\mathfrak{m}}$.

3.3 Forward Pre-Analysis

The original characterization of existential recurrent set (given in Lemma 2.5) suggests to initialize the computation with just the set of non-error non-final states and then perform the analysis by iterating the predecessors transformer (pre). But to better direct the search for an existential recurrent set, we first perform a forward pre-analysis of a program and find a(n over-approximation of the) set of non-error states reachable from some initial state. Then, backward analysis will be searching for an existential recurrent set below this set of reachable states. We observed that this approach works better in practice, and this is consistent with observations of other researchers who note that a combination of backward and forward analyses is known to be more precise

than just, e.g., backward analysis [CC99]. Intuitively, the analysis in a numeric abstract domain infers an existential recurrent set by collecting constraints (i.e., linear inequalities) from across the program and combining them. Forward analysis propagates the constraints forwards, from the locations where they are introduced, to the locations where they are important for non-termination. Backward analysis works similarly, but propagates the constraints backwards. It turns out that some important constraints are better propagated by forward analysis than by backward analysis, e.g. the loop and branching conditions (formulas from assumption statements). Thus, pure backward analysis below the set of non-error non final states might not be able to make use of loop and branching conditions and infer an existential recurrent set.

Thus, we start by performing a standard forward pre-analysis of the whole program $\mathbb{P}$ to find an approximation of the set of non-error reachable program states $F \in \mathbb{D}_\sharp$. One way to do so it the following. Let F_ε be the stable limit of the sequence $\{f_i\}_{i\geq 0}$ where

$$\begin{aligned} f_0 &= \{\lfloor_\vdash \mapsto \top_{\mathrm{mp}}; l \neq \lfloor_\vdash \mapsto \bot_{\mathrm{mp}}\} \\ f_i &= f_{i-1} \triangledown_\sharp (f_{i-1} \sqcup_\sharp \mathrm{post}^\sharp(\mathbb{P}, f_{i-1})), \text{ for } i \geq 1 \end{aligned} \tag{3.1}$$

This computes a standard forward analysis in the domain $\mathbb{D}_\sharp$. Then, take the resulting approximation $F \sqsubseteq_\sharp F_\varepsilon$ to be the greatest non-error element below F_ε.

Another way is to introduce an abstract operation $\mathrm{post}^{\mathfrak{m}}_{\mathrm{ok}}$ that over-approximates taking non-error successors:

$$\gamma_{\mathfrak{m}}(\mathrm{post}^{\mathfrak{m}}_{\mathrm{ok}}(C, a)) \supseteq \mathrm{post}(T_{\mathbb{M}}(C), \gamma_{\mathfrak{m}}(a)) \setminus \{\varepsilon\}$$

and then compute a forward analysis in the domain of non-error abstract states (if this domain can be constructed). This second way can be useful when the abstract domain was built using the "error top" construction, as described in Section 2.3.3. Then, for an unsafe program, standard forward analysis is likely to lose information about which non-error states are reachable, while the analysis based on the computation of $\mathrm{post}^{\mathfrak{m}}_{\mathrm{ok}}$ will not be affected.

Example 3.1 (Non-Deterministic Branches)**.** Let us consider the program fragment in Fig. 3.3, which will be our running example for this chapter. We call it a fragment here, as it only shows a loop, without the preceding or subsequent parts of a program, and,

e.g., there is no edge that exits the loop. One can expect that this is a part of a larger program, which, e.g., would have an edge[3] going from location l_1 labelled with an assumption statement $[x < 0 \vee x > 100]$ that leads outside of the loop. In this case, we assume that it is only in this loop where non-terminating executions can arise, and hence the rest of the program is not interesting for the discussion. In particular, we are only interested in the edges that allow the execution to stay inside the loop, and all the edges that lead outside are not relevant and do not affect the computation of a recurrent set.

For this fragment, the set of locations is $\mathbb{L} = \{l_1, \cdots, l_4\}$ and the initial location is $l_\vdash = l_1$. The program does not have a final location and thus is a strongly connected fragment of a larger program. Note how we cannot have multiple edges from l_2 to l_1, and we use locations l_3 and l_4 to work around that (for the edges displayed without a label, we assume the label `skip`).

Now, let us see what forward pre-analysis will produce for this fragment. Informally, we can see that we do not know which states enter the loop, but at location l_2 the states need to satisfy the condition $0 \le x \le 100$. Thus, the pre-analysis may produce the element $F \in \mathbb{D}_\sharp$, s.t.

$$F(l_1) = \langle\rangle \mapsto \top$$
$$F(l_2) = \langle\rangle \mapsto (0 \le x \le 100)$$
$$F(l_3) = \langle\rangle \mapsto (1 \le x \le 101)$$
$$F(l_4) = \langle\rangle \mapsto (-1 \le x \le 99)$$

In our prototype implementation, forward analysis does not perform trace partitioning, thus, for every location, F has one partition labelled with the top element of the path domain (recall that in our instantiation of path domain, the empty sequence $\langle\rangle$ is the top element).

[3] For a numeric program, it may actually be convenient to require that an assumption formula is a conjunction of linear inequalities and thus can be represented by a single element of polyhedral domain. In this case, we will create not one but two edges leaving the loop from location l_1, one labelled by $[x < 0]$ and another – by $[x > 100]$. Example 3.4 demonstrates that this makes our analysis more precise.

3.4 Backward Analysis For a Candidate

Next, we perform the main step of the analysis – backward analysis to find candidate existential recurrent sets (possibly, over-approximated). We perform this analysis separately for every strongly connected sub-program $\mathbb{P}_s$ that represents a loop of the original program $\mathbb{P}$. More formally, we perform the analysis for every *strongly connected component* [Tar72] $\mathbb{P}_s = (\mathbb{L}_s, \mathfrak{l}_{s\vdash}, \mathbb{E}_s, \mathfrak{c}|_{\mathbb{E}_s})$, where

- the subprogram $\mathbb{P}_s$ is strongly connected and no subprogram that properly contains $\mathbb{P}_s$ is strongly connected;
- $\mathbb{L}_s \subseteq \mathbb{L}$;
- $\mathbb{E}_s = (\mathbb{L}_s \times \mathbb{L}_s) \cap \mathbb{E}$;
- additionally, $|\mathbb{L}_s| > 1$ or $(\mathfrak{l}_{s\vdash}, \mathfrak{l}_{s\vdash}) \in \mathbb{E}_s$ (i.e., trivial strongly connected components are disregarded, and the component should represents a loop in the program);
- $\mathfrak{c}|_{\mathbb{E}_s}$ is the restriction of $\mathfrak{c}$ to the edges of $\mathbb{P}_s$;
- $\mathfrak{l}_{s\vdash} \in \mathbb{L}_s$ is the *head* of the strongly connected component, which is usually selected as the first location of the component encountered in $\mathbb{P}$ by a depth-first search.

Note that since $\mathbb{P}_s$ is strongly connected, it does not have final locations.

We can restrict the notion of successors to a sub-program. For $l \in \mathbb{L}_s$,

$$\mathrm{succ}(l)|_{\mathbb{P}_s} = \{l' \in \mathbb{L}_s \mid (l, l') \in \mathbb{E}_s\}$$

Lemma 3.1. An existential recurrent set of a sub-program $\mathbb{P}_s$ is an existential recurrent set of the original program $\mathbb{P}$.

Proof. The proof is straightforward and follows from that a non-terminating execution postfix of a subprogram is also a non-terminating execution postfix of the original program. □

For every strongly connected sub-program $\mathbb{P}_s$, we find a candidate existential recurrent set $W_s \in \mathbb{D}_\sharp$ as the stable limit of the sequence of elements $\{w_i\}_{i\geq 0}$ that approx-

imates non-termination analysis below F, s.t.

$$\begin{aligned} w_0 &= F|_{\mathbb{L}_s} \\ w_i &= w_{i-1} \mathbin{\underline{\triangledown}_\sharp} (w_{i-1} \sqcap_\sharp \mathrm{pre}^\sharp(\mathbb{P}_s, w_{i-1})), \text{ for } i \geq 1 \end{aligned} \tag{3.2}$$

where $F|_{\mathbb{L}_s}$ is the restriction of F to the locations of $\mathbb{P}_s$:

$$\begin{aligned} F|_{\mathbb{L}_s} &= F(l), \text{ for } l \in \mathbb{L}_s \\ F|_{\mathbb{L}_s} &\text{ is not defined for } l \notin \mathbb{L}_s \end{aligned}$$

Although formally an element of $\mathbb{D}_\sharp$ concretizes to a set of traces, we can think that W_s represents a candidate existential recurrent set:

$$\alpha_{\mathfrak{s}}(\gamma_\sharp(W_s)) = \{(l, m) \in \mathbb{S} \mid \exists q \in \mathbb{D}_{\mathfrak{p}}.\ m \in \gamma_{\mathfrak{m}}(W_s(l)(q))\}$$

Since we use over-approximate operations (join, backward transformers) to compute W_s, and hence the computation may not under-approximate non-termination analysis and W_s might not represent a genuine existential recurrent set. In the next analysis step, we will produce a refined element $R_s \sqsubseteq_\sharp W_s$ representing a genuine existential recurrent set.

While this is hidden by succinctness of the definition of W_s, in practice, trace partitioning is important for inferring a good candidate. We observed that for many imperative programs, non-terminating executions take a specific path through the loop. When we perform backward analysis with trace partitioning, abstract memory states in W_s are partitioned by the path through the loop that the program run would take from them. If the path domain is expressive enough, s.t. (states, from which exist) non-terminating semi-executions get collected in separate partitions, the analysis is likely to find a good candidate.

Example 3.1 (Non-Deterministic Branches, continued)**.** Let us return to the program fragment in Fig 3.3. One can see that in every iteration of the loop, the execution makes a non-deterministic choice: whether to increment or decrement the variable x. For this fragment, a non-terminating execution in every iteration needs to make the choice depending on the current value of x, so that it does not go outside the range $[0, 100]$. This can actually be captured by our path domain, and for simplicity let us assume that

an element of the path domain only remembers one next branching choice. This will be enough in this example[4].

Now let us trace a few iterations of the backward analysis and show how it computes the candidate recurrent set W_s. We initialize the computation with the result of forward pre-analysis. Recall, that earlier we established that forward pre-analysis will produce an element $F \in \mathbb{D}_\sharp$, s.t. (note that in our instantiation of path domain, the empty sequence $\langle\rangle$ is the top element)

$$F(l_1) = \langle\rangle \mapsto \top$$
$$F(l_2) = \langle\rangle \mapsto (0 \le x \le 100)$$
$$F(l_3) = \langle\rangle \mapsto (1 \le x \le 101)$$
$$F(l_4) = \langle\rangle \mapsto (-1 \le x \le 99)$$

Thus, we take the initial approximation of W_s to be F.

In this case, a reasonable iteration order (the order in which we update the entries of W_s) is $(l_1, l_3, l_4, l_2)*$. Thus, we compute:

$$W_s(l_1)_1 = W_s(l_1)_0 \sqcap_{\text{mp}} \text{pre}^{\text{mp}}((l_1, l_2), W_s(l_2)_0) = \langle\rangle \mapsto (0 \le x \le 100)$$
$$W_s(l_3)_1 = W_s(l_3)_0 \sqcap_{\text{mp}} \text{pre}^{\text{mp}}((l_1, l_3), W_s(l_1)_1) = \langle\rangle \mapsto (1 \le x \le 100)$$
$$W_s(l_4)_1 = W_s(l_4)_0 \sqcap_{\text{mp}} \text{pre}^{\text{mp}}((l_1, l_4), W_s(l_1)_1) = \langle\rangle \mapsto (0 \le x \le 99)$$
$$W_s(l_2)_1 = W_s(l_2)_0 \sqcap_{\text{mp}} \left(\text{pre}^{\text{mp}}((l_2, l_3), W_s(l_3)_1) \sqcup_{\text{mp}} \text{pre}^{\text{mp}}((l_2, l_4), W_s(l_4)_1)\right)$$
$$= \{\langle(l_2, l_3)\rangle \mapsto (0 \le x \le 99); \langle(l_2, l_4)\rangle \mapsto (1 \le x \le 100)\}$$

Note how the join operation created two partitions corresponding to two branching choices a location l_2. We continue.

$$W_s(l_1)_2 = W_s(l_1)_1 \sqcap_{\text{mp}} \text{pre}^{\text{mp}}((l_1, l_2), W_s(l_2)_1)$$
$$= \{\langle(l_2, l_3)\rangle \mapsto (0 \le x \le 99); \langle(l_2, l_4)\rangle \mapsto (1 \le x \le 100)\}$$
$$W_s(l_3)_2 = W_s(l_3)_1 \sqcap_{\text{mp}} \text{pre}^{\text{mp}}((l_1, l_3), W_s(l_1)_2)$$
$$= \{\langle(l_2, l_3)\rangle \mapsto (1 \le x \le 99); \langle(l_2, l_4)\rangle \mapsto (1 \le x \le 100)\}$$
$$W_s(l_4)_2 = W_s(l_4)_1 \sqcap_{\text{mp}} \text{pre}^{\text{mp}}((l_1, l_4), W_s(l_1)_2)$$

[4]The actual implementation of this analysis uses a different definition of bounded element, the one offered in Section 3.2.

$$= \{\langle (l_2, l_3)\rangle \mapsto (0 \le x \le 99); \langle (l_2, l_4)\rangle \mapsto (1 \le x \le 99)\}$$

Note how the partitions in locations l_3 and l_4 correspond to the branching choices in the next iteration of the loop. We now proceed to location l_2:

$$W_s(l_2)_2 = W_s(l_2)_1 \sqcap_{\mathfrak{mp}} \left(\mathrm{pre}^{\mathfrak{mp}}((l_2, l_3), W_s(l_3)_2) \sqcup_{\mathfrak{mp}} \mathrm{pre}^{\mathfrak{mp}}((l_2, l_4), W_s(l_4)_2)\right)$$

In this case,

$$\begin{aligned}
\mathrm{pre}^{\mathfrak{mp}}((l_2, l_3), W_s(l_3)_2) = \mathrm{reduce}\big(\{\mathfrak{b}_1(\langle (l_2, l_3), (l_2, l_3)\rangle) \mapsto \mathrm{pre}^{\mathfrak{m}}(x \leftarrow x+1, 1 \le x \le 99),\\
\mathfrak{b}_1(\langle (l_2, l_3), (l_2, l_4)\rangle) \mapsto \mathrm{pre}^{\mathfrak{m}}(x \leftarrow x+1, 1 \le x \le 100)\}\big)\\
= \langle (l_2, l_3)\rangle \mapsto \mathrm{pre}^{\mathfrak{m}}(x \leftarrow x+1, 1 \le x \le 99) \sqcup_{\mathfrak{m}}\\
\mathrm{pre}^{\mathfrak{m}}(x \leftarrow x+1, 1 \le x \le 100)\\
= \langle (l_2, l_3)\rangle \mapsto (0 \le x \le 99)
\end{aligned}$$

What happens here is that (l_2, l_3) is a branching choice. Since we chose to only keep at most one future branching choice in a path domain element, the predecessor operation for the edge (l_2, l_3) produces only one abstract partition, and the corresponding abstract memory state is the join of the predecessors of the abstract memory states in $W_s(l_3)_2$. Similarly,

$$\mathrm{pre}^{\mathfrak{mp}}((l_2, l_4), W_s(l_4)_2) = \langle (l_2, l_3)\rangle \mapsto (1 \le x \le 100)$$

This is the kind of over-approximating operation that we allow in the analysis. In general, this may result in an unsound candidate recurrent set and that is why we need to check the candidate later. Now,

$$W_s(l_2)_2 = \{\langle (l_2, l_3)\rangle \mapsto (0 \le x \le 99); \langle (l_2, l_4)\rangle \mapsto (1 \le x \le 100)\} = W_s(l_2)_1$$

The entry of W_s for location l_2 has stabilized, and further computation will not make

any more updates to the candidate. Thus, the resulting candidate is:

$$W_s(l_1) = \{\langle(l_2, l_3)\rangle \mapsto (0 \le x \le 99); \langle(l_2, l_4)\rangle \mapsto (1 \le x \le 100)\}$$
$$W_s(l_2) = W_s(l_1)$$
$$W_s(l_3) = \{\langle(l_2, l_3)\rangle \mapsto (1 \le x \le 99); \langle(l_2, l_4)\rangle \mapsto (1 \le x \le 100)\}$$
$$W_s(l_4) = \{\langle(l_2, l_3)\rangle \mapsto (0 \le x \le 99); \langle(l_2, l_4)\rangle \mapsto (1 \le x \le 99)\}$$

This can be interpreted as follows. If the execution is at location l_1 and, as the next branching choice, is going to increment x (by taking the edge (l_2, l_3)), then, for the execution to not leave the loop, it must be that $0 \le x \le 99$. Indeed, if $x < 0$, the execution will not enter the loop, and if $x > 99$, the execution will exit the loop after incrementing x. Similarly, if the execution is going to decrement x, it must be that $1 \le x \le 100$. That is, if the execution is at location l_1, and $0 \le x \le 100$, there exists a branching choice at location l_2 that keeps x in range $[0, 100]$. This way we can construct a non-terminating execution. By this argument, W_s represents a genuine recurrent set.

3.5 Checking and Refining a Candidate

Approximate backward analysis for every strongly connected component $\mathbb{P}_s$ of the original program produces an element $W_s \in \mathbb{D}_\sharp$, which represents a candidate existential recurrent set. We use over-approximate operations (join, backward transformers) to compute W_s, and hence the computation may not under-approximate non-termination analysis and W_s might not represent a genuine existential recurrent set. In Example 3.1, we were able to produce a genuine recurrent set with backward analysis, but this is just a lucky coincidence.

In general, we have to refine W_s to a (possibly, bottom) element $R_s \sqsubseteq_\sharp W_s$ representing a genuine existential recurrent set of $\mathbb{P}_s$ and hence of the original program $\mathbb{P}$. That is, we produce such R_s that $\forall s \in \alpha_{\mathbb{s}}(\gamma_\sharp(R_s)).\ \exists s' \in \alpha_{\mathbb{s}}(\gamma_\sharp(R_s)).\ (s, s') \in T_{\mathbb{S}}(\mathbb{P}_s)$. To do so, we define a predicate CONT, s.t. for an abstract memory state $a \in \mathbb{D}_{\mathbb{m}}$, a set of abstract memory states $A \subseteq \mathbb{D}_{\mathbb{m}}$, and an atomic statement $C \in \mathbb{A}$, if $\mathrm{CONT}(a, C, A)$ holds (we say that the run of the program can continue from a to A through C) then $\forall m \in \gamma_{\mathbb{m}}(a).\ \exists m' \in \gamma_{\mathbb{m}}(A).\ (m, m') \in T_{\mathbb{M}}(C)$. We define CONT separately for different kinds of atomic statements.

For the memory abstract domain, let us introduce an additional *coverage* operation $\sqsubseteq^+_{\mathbb{m}}$ that generalizes abstract order. For an abstract memory state $a \in \mathbb{D}_{\mathbb{m}}$ and a set $A \subseteq \mathbb{D}_{\mathbb{m}}$, it should be that if $a \sqsubseteq^+_{\mathbb{m}} A$ (we say that a is *covered* by A) then $\gamma_{\mathbb{m}}(a) \subseteq \gamma_{\mathbb{m}}(A)$. For an arbitrary domain, coverage can be defined via the Hoare order:

$$a \sqsubseteq^+_{\mathbb{m}} A \text{ iff } \exists a' \in A.\ a \sqsubseteq_{\mathbb{m}} a'.$$

For a numeric domain, it is usually possible to define a more precise coverage operation. For example, a popular implementation of a family of numeric domains – Parma Polyhedra Library [BHZ08] – defines a specialized coverage operation for finite sets of convex polyhedra.

We define CONT as follows, using operations that are standard in program analysis. For $a \in \mathbb{D}_{\mathbb{m}}$, $A \subseteq \mathbb{D}_{\mathbb{m}}$,

(i) For the passive statement `skip`,

$$\text{CONT}(a, \texttt{skip}, A) \equiv a \sqsubseteq^+_{\mathbb{m}} A$$

Indeed, if $a \sqsubseteq^+_{\mathbb{m}} A$ then $\gamma_{\mathbb{m}}(a) \subseteq \gamma_{\mathbb{m}}(A)$, and hence it holds that $\forall m \in \gamma_{\mathbb{m}}(a).\ \exists m' = m \in \gamma_{\mathbb{m}}(A).\ (m, m') = (m, m) \in T_{\mathbb{M}}(\texttt{skip})$.

(ii) For an assumption statement $[\theta]$,

$$\text{CONT}(a, [\theta], A) \equiv \left(\text{eval}^{\mathbb{m}}(\theta, a) = 1\right) \wedge a \sqsubseteq^+_{\mathbb{m}} A$$

Indeed, if $\text{eval}^{\mathbb{m}}(\theta, a) = 1$, then $\gamma_{\mathbb{m}}(a) \subseteq [\![\theta]\!]$, and if additionally $a \sqsubseteq^+_{\mathbb{m}} A$ then $\forall m \in \gamma_{\mathbb{m}}(a).\ \exists m' = m \in \gamma_{\mathbb{m}}(A).\ (m, m') = (m, m) \in T_{\mathbb{M}}([\theta])$.

(iii) For a nondeterministic assignment $x \leftarrow *$, we use the fact that in many numeric domains (including the polyhedral domain) the pre-condition of $x \leftarrow *$ can be computed precisely (via a *cylindrification* operation [HMT71]). That is, for $a \in \mathbb{D}_{\mathbb{m}}$, $\gamma_{\mathbb{m}}(\text{pre}^{\mathbb{m}}(x \leftarrow *, a)) = \{m \in \mathbb{M} \mid \exists m' \in \gamma_{\mathbb{m}}(a).\ (m, m') \in T_{\mathbb{M}}(x \leftarrow *)\}$. In this case,

$$\text{CONT}(a, x \leftarrow *, A) \equiv a \sqsubseteq^+_{\mathbb{m}} \{\text{pre}^{\mathbb{m}}(x \leftarrow *, a') \mid a' \in A\}$$

(iv) Finally, for every other atomic statement C with left-total input-output relation $T_{\mathbb{M}}(C)$ (e.g., a deterministic assignment),

$$\mathrm{CONT}(a, C, A) \equiv \mathrm{post}^{\mathfrak{m}}(C, a) \sqsubseteq^{+}_{\mathfrak{m}} A$$

Indeed, in this case $\gamma_{\mathfrak{m}}(A) \supseteq \gamma_{\mathfrak{m}}(\mathrm{post}^{\mathfrak{m}}(C, a)) \supseteq \mathrm{post}(C, \gamma_{\mathfrak{m}}(a))$. Since additionally, $T_{\mathbb{M}}(C)$ is left-total then for every $m \in \gamma_{\mathfrak{m}}(a)$. $\exists m' \in \gamma_{\mathfrak{m}}(A)$. $(m, m') \in T_{\mathbb{M}}(C)$.

Another way to look at it is that (iv) represents a general case that allows handling atomic statements with left-total input-output relations. Then, we specialize CONT for non-deterministic statements and for statements with non-left-total input-output relations. Case (iii) specializes CONT for non-deterministic assignments. It allows us to detect a situation where there exists a *specific* non-deterministic choice (i.e., a specific new value of a variable) that keeps the execution inside the existential recurrent set. Case (ii) specializes CONT for assumption statements (with non-left-total input-output relations). By extending the definition of CONT, we can extend our analysis to support more kinds of atomic statements.

Theorem 3.1. Let $R_s \in \mathbb{D}_\sharp$ be an element of $\mathbb{D}_\sharp$ and $\mathbb{P}_s$ be a sub-program. Let it be that for every location $l \in \mathbb{L}_s$, abstract path $q \in \mathbb{D}_{\mathfrak{p}}$, and an abstract memory state $a \in \mathbb{D}_{\mathfrak{m}}$, s.t. $R_s(l)(q) = a$, there exists a successor location $l' \in \mathrm{succ}(l)|_{\mathbb{P}_s}$, s.t.

$$\mathrm{CONT}(a, \mathfrak{c}(l, l'), \{a' \mid \exists q' \in \mathbb{D}_{\mathfrak{p}}.\ a' = R_s(l')(q')\})$$

Then, R_s represents an existential recurrent set of the sub-program $\mathbb{P}_s$ and hence the whole program $\mathbb{P}$.

Proof. The proof is a straightforward application of the definitions of CONT and $T_{\mathbb{S}}$. Intuitively, if $R_s \in \mathbb{D}_\sharp$ satisfies the condition of the lemma, from every program state in $\alpha_{\mathfrak{s}}(\gamma_\sharp(R_s))$ we can form a non-terminating semi-execution that only visits the elements of $\alpha_{\mathfrak{s}}(\gamma_\sharp(R_s))$ – by executing the statements of $\mathbb{P}_s$ in a specific order, i.e., always choosing an edge for which the predicate CONT holds. □

Informally, Theorem 3.1, states the following. For every abstract memory state in R_s, there should be an edge (outgoing from the corresponding location), s.t. taking this edge from *every* corresponding concrete state keeps the execution inside the existential

recurrent set. Note that we do not need to consider the case where for a given abstract state, for different concrete states, the execution needs to take different outgoing edges. This is because ouf our path domain. During backward analysis, at branching points, abstract states will always be partitioned according to which branch they are going to take in a non-terminating execution.

Refinement Step In the refinement step, we start with an element $W_s \in \mathbb{D}_\sharp$ produced by the backward analysis, and from every location $l \in \mathbb{L}_s$, we repeatedly exclude the tuples $(q, a) \in W_s(l)$ that violate the condition of Theorem 3.1. More formally, if we repeatedly try to find a location $l \in \mathbb{L}_s$ and an abstract path $q \in \mathbb{D}_\mathfrak{p}$ for which we cannot find a successor location $l' \in \text{succ}(l)|_{\mathbb{P}_s}$, s.t. $\text{CONT}(a, \mathfrak{c}(l, l'), \{a' \mid \exists q' \in \mathbb{D}_\mathfrak{p}.\ a' = R_s(l')(q')\})$. After finding such l and q, we remove the corresponding tuples from W_s. Eventually, we arrive at an element $R_s \sqsubseteq_\sharp W_s$ that satisfies Theorem 3.1 and hence, represents an existential recurrent set. Note that the refinement step that we implement in this chapter is coarse. For some disjunct $(q, a) \in W_s(l)$, we either keep it unchanged or remove it as a whole. In particular, an empty set is trivially existentially recurrent, and it is still sound to produce $R_s = \bot_\sharp$. We believe that is acceptable. The *form* of the existential recurrent set in our current implementation is inferred by the backward and forward analysis steps, and the refinement step is mainly designed to ensure soundness.

As a direction for future research, we note that the analysis may benefit from the ability to modify individual disjuncts during refinement. That is, when we find in W_s an abstract state a (or more formally, a location l and a path q, s.t. $W_s(l)(q) = a$) that violates the condition of Theorem 3.1, instead of completely removing a from W_s, we could first try to find $a' \sqsubseteq_\mathfrak{m} a$, s.t. a' does not violate the condition of the theorem.

Finally, we note that in principle, our procedure does not have to be applied to a strongly connected sub-program. It can be applied to the original program with final locations excluded, and this way we can prove non-termination: if $R_s(\mathfrak{l}_\vdash)$ is defined and $\gamma_\sharp(R_s(\mathfrak{l}_\vdash)) \neq \varnothing$, then there exists at least one non-terminating program execution. But so far, we had little practical success with this approach. While our path domain $\mathbb{D}_\mathfrak{p}$ is sufficient to capture some non-terminating paths through loops, it is not expressive enough to capture non-terminating paths through the whole program (unless it consists of a single loop and a sequential stem). Thus, for practical reasons, we search

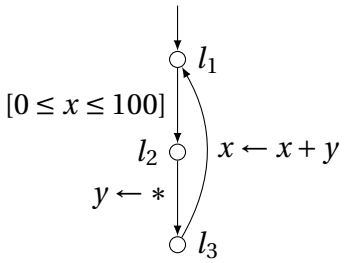


Figure 3.4: Loop that assigns a non-deterministic value to a variable in every iteration.

for existential recurrent sets of individual loops and assume that a reachability analysis will complete the non-termination proof, if necessary.

3.6 Examples

In this section, we present additional numeric examples that demonstrate how different components of the analysis (trace partitioning, CONT, lower widening) are important for different kinds of non-terminating behaviors. In all examples, we assume that program variables take integer values[5] and there is no bound on their values (the latter is a common assumption in similar research on numeric programs, e.g., [LQC15]). Also, we assume that the analysis uses the polyhedral domain (i.e., sets of linear inequalities) to represent memory states.

Example 3.2 (Non-Deterministic Assignment in the Loop)**.** Fig. 3.4 shows a loop that in every iteration, first assigns a non-deterministic value to y and then adds it to x. Intuitively, if at location l_1 x is in range $[0,100]$, then for the edge (l_2,l_3), there is always a choice of y, s.t. $x+y$ is still in the range $[0,100]$. In this way, we can construct a non-terminating execution.

The way, in which we define the predicate CONT for non-deterministic assignments, allows us to handle such cases. The first two steps (pre-analysis and backward analysis) yield the candidate recurrent set W_s, s.t.

$$W_s(l_1) = \{\langle\rangle \mapsto (0 \le x \le 100)\}$$

$$W_s(l_2) = W_s(l_1)$$

$$W_s(l_3) = \{\langle\rangle \mapsto (0 \le x \le 100 \wedge 0 \le x+y \le 100)\}$$

[5] In particular, sound analysis of floating point operations is not among our goals.

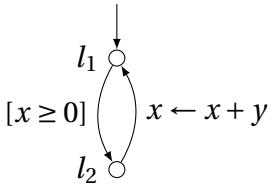


Figure 3.5: Loop that requires a specific range of y for non-termination.

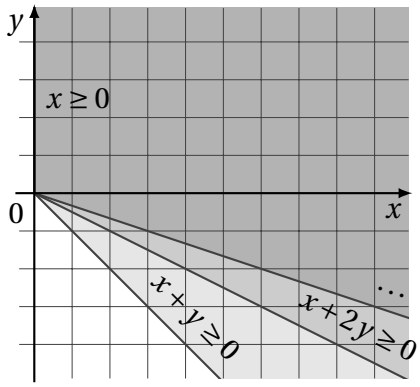


Figure 3.6: Illustration of the descending chain $\{x \geq 0 \wedge x + jy \geq 0\}_{j \geq 0}$.

We show that W_s satisfies Theorem 3.1 and thus represents a genuine recurrent set. For location l_1, the successor location is l_2, and $(0 \leq x \leq 100)$ satisfies the memory-state formula of the assumption statement that labels (l_1, l_2). That is, for every state at location l_1 with $0 \leq x \leq 100$, we will stay in the recurrent set after executing the assumption statement. This corresponds to case (ii) of the predicate CONT. For location l_2, the successor location is l_3 and $\mathfrak{c}(l_2, l_3)$ is the non-deterministic assignment $y \leftarrow *$. Note that for every value of x it is possible to choose a value of y, s.t. $0 \leq x+y \leq 100$ holds. Or, more formally, $\text{pre}^{\mathfrak{m}}(y \leftarrow *, (0 \leq x \leq 100 \wedge 0 \leq x+y \leq 100)) = (0 \leq x \leq 100)$ which corresponds to case (iii) of the predicate CONT. Finally, for location l_3, the successor location is l_1 and $\mathfrak{c}(l_3, l_1)$ is $x \leftarrow x+y$. Also, $\text{post}^{\mathfrak{m}}(x \leftarrow x+y, (0 \leq x \leq 100 \wedge 0 \leq x+y \leq 100)) = (0 \leq x-y \leq 100 \wedge 0 \leq x \leq 100) \sqsubseteq (0 \leq x \leq 100)$ which corresponds to case (iv) of the predicate CONT. Therefore, W_s represents a genuine recurrent set, and the final step of the analysis yields $R_s = W_s$.

Example 3.3 (Non-Deterministic Assignment Before the Loop)**.** Fig. 3.5 shows a loop that in every iteration adds y to x. Both x and y are not initialized before the loop, and are thus assumed to take non-deterministic values. If at location l_1, $x \geq 0$ and $y \geq 0$, it is possible to continue the execution forever. Let us see how the constraint $y \geq 0$ can be inferred with lower widening. For this program, the pre-analysis produces the invariant F, s.t.

$$F(l_1) = \{\langle\rangle \mapsto \top\}$$

$$F(l_2) = \{\langle\rangle \mapsto x \geq 0\}$$

Then, consider a sequence of approximants $\{w_i\}_{i\geq 0}$ where

$$w_0 = F$$
$$w_i = w_{i-1} \sqcap_\sharp \mathrm{pre}^\sharp(\mathbb{P}, w_{i-1}), \text{ for } i \geq 1$$

which corresponds to running the backward analysis without lower widening. Then, we will observe that the i-th approximant at location l_1 represents the condition that ensures that the execution will make at least i iterations through the loop. For $i \geq 0$, let $w'_i = w_i(l_1)(\langle\rangle)$. Then

$$w'_0 = \top$$
$$w'_1 = x \geq 0$$
$$w'_2 = (x \geq 0 \wedge x+y \geq 0)$$
$$w'_3 = (x \geq 0 \wedge x+2y \geq 0)$$
$$w'_4 = (x \geq 0 \wedge x+3y \geq 0)$$
$$\cdots$$

That is, for $i \geq 1$, $w'_i = (x \geq 0 \wedge x + iy \geq 0)$ (a polyhedron with a "rotating" constraint, see Fig. 3.6), and we would like a lower widening technique that would produce an extrapolated polyhedron $(x \geq 0 \wedge y \geq 0)$ which is the limit of the chain $\{w'_i\}_{i\geq 0}$. Notice how this limit is below w'_i for every $i \geq 0$. This explains why we use lower widening (and not, e.g., narrowing) to ensure convergence of the backward analysis. Here, we use lower widening as proposed by A. Miné [Min13]. Intuitively, it works by retaining stable generators (which can be seen as dual to standard widening that retains stable constraints). Additionally, we use widening delay of 2 and a technique of threshold rays (also described in [Min13]), adding the coordinate vectors and their negations to the set of thresholds. Alternatively, instead of using threshold rays, one could adapt to lower widening the technique of evolving rays [Bag+05]. This allows the backward analysis to produce the extrapolated polyhedron $(x \geq 0 \wedge y \geq 0)$. Eventually, backward analysis produces the candidate W_s where

$$W_s(l_1) = \{\langle\rangle \mapsto (x \geq 0 \wedge y \geq 0)\}$$
$$W_s(l_2) = W_s(l_1)$$

```
1 | if (a < b)
2 |   swap(a, b)
3 | while (a ≠ b) {
4 |   t ← a − b;
5 |   a ← b;
6 |   b ← t;
7 | }
```

Figure 3.7: GCD algorithm with an introduced bug.

W_s represents a genuine recurrent set, and the final (refinement) step of the analysis yields $R_s = W_s$.

Example 3.4. This example is a program "GCD" from the test set of Invel[6] [VR08]. The program given in pseudocode in Fig. 3.7 is based on the basic algorithm that computes the greatest common divisor of two numbers: a and b – but has an introduced bug that produces non-terminating behaviors. For the loop in this program, our analysis (with $k = 2$) is able to show that if at line 3, it is the case that $(a > b \wedge a > 2b)$ or $(b > a \wedge 2b > a)$, the execution will never terminate and will alternate between these two regions. This example demonstrates how the interaction between the components of the analysis allows finding non-trivial non-terminating behaviors. In a program graph, the condition $a \neq b$ will be represented by a pair of edges, labelled by assumption statements: $[a > b]$ and $[a < b]$. Thus, these assumption statements become branching choices at line 3. Then, the path domain (with k at least 2) allows the analysis to distinguish the executions that alternate between these two assumption statements for the first k loop iterations. By doing numeric reasoning, one can check that there exist non-terminating executions that alternate between the two assumption statements indefinitely.

The example also demonstrates a non-trivial refinement step. At line 3, backwards analysis actually yields two additional disjuncts, one of those being $(a > b \wedge 2b > a \wedge 3b - a > 4)$. These are the states that take the branching choice $[a > b]$ for at least two first loop iterations. But from some of the concrete states in the disjunct, e.g., $(a = 6, b = 4)$, the loop eventually terminates. As currently implemented, the refinement

[6] `http://www.key-project.org/nonTermination/`, last accessed in May 2016.

step has to remove the whole disjunct from the final result.

Finally, note how for this example, recurrent set cannot be represented by a single convex polyhedron (per program location). Our approach allows to keep multiple polyhedra per location, corresponding to different abstract paths.

To summarize, the components of the analysis are responsible for handling different features of non-terminating executions. Trace partitioning allows predicting paths that non-terminating executions take; predicate CONT deals with non-deterministic statements in a loop; lower widening infers the required values of variables that are non-deterministically set outside of a loop.

3.7 Related Work

The idea of proving non-termination by looking at paths of a certain form appears in multiple existing works. We note though, that usually the authors of previous work were interested in proving non-termination, while we are solving a sub-problem of that (finding an existential recurrent set). It is still valid in most cases to compare our analysis to existing approaches and tools. Search for a non-terminating execution postfix is still a substantial (arguably, the most important) part of a non-termination proof.

An early analysis by Gupta et al. [Gup+08] enumerates symbolic executions of a program and tries to find one that represents a lasso-shaped one: a non-terminating execution a certain sequence of instructions is executed infinitely often. The analysis is formulated for linear programs and uses Farkas' lemma to produce the proof.

The tool called Ultimate Büchi Automizer [HHP14; Hei+16] decomposes the original program into a set of lasso-programs (a lasso programs in a sequential stem followed by a loop with no branches) to separately infer termination or non-termination [LH14] arguments for them.

The tool AProVE [Gie+14] implements a range of techniques. The one that is interesting in the context of this chapter is described in paper [Bro+11]. For every loop, it analyses a set of paths through it and produces a formula that is unsatisfiable if there is a set of states that cannot be escaped by following these paths. The formula is then passed on to an SMT solver.

Similarly to these analyses, our approach tries to identify a path through a loop

that a non-terminating execution takes, but uses trace partitioning for that. This does not have to be the same path segment repeated infinitely often, but may also be an alternation of different segments[7]. We see a strength of our approach in that it is parameterized by a path domain. That is, the partitioning scheme can be improved in future work and/or specialized for different classes of programs.

Chen et al. [Che+14] also use a combination of forward and backward analysis, but in a different way. With forward analysis, they identify terminating abstract traces; then using backward analysis over a single trace, they identify, how the program can be restricted (by adding assumption statements) to remove this trace. By repeating this process, they may be able to obtain a program without any terminating executions. Then, they try show that the restricted program has at least one execution (which is non-terminating by construction). This final step in their approach is actually similar to the refinement step of our analysis.

A distinctive approach is implemented in a tool called E-HSF [BPR13]. It allows the user to specify the semantics of a program and the verified properties in the form of $\forall\exists$ quantified Horn clauses. In particular, the specification language allows to assert the existence of different kinds of recurrent sets (to specify liveness properties, the specification may include well-foundedness assertions). To our knowledge, the implementation is targeted at numeric programs and relies on Farkas' lemma (although the general approach of using horn clauses as a specification language is more general).

There is also a number of approaches that can infer universal recurrent sets for programs (or prove non-termination by finding a reachable universal recurrent set). It is actually reasonable to compare them with the analysis of this chapter, as for a certain class of deterministic programs, the notions of universal and existential recurrent sets coincide. We do not do it here though, but in Chapters 4 and 5.

Finally, [MR05] presents a different formalization of trace partitioning (in the context of standard forward analysis). In particular, the authors describe a more expressive path domain that, e.g., can represent paths that take a finite number of iterations through a loop. Adapting such a domain to our analysis can be a topic for future work.

[7] This is what is called *non-periodic non-termination*. AProVE can also find such behaviours.

3.8 Chapter Conclusion and Future Work

In this chapter, we proposed an analysis that finds existential recurrent sets of the loops in (numeric) imperative programs. The analysis is based on the combination of forward and backward abstract interpretation and an important technique that we use is trace partitioning. To our knowledge, this is the first application of trace partitioning to backward analysis.

In Chapter 5 we will see that the implementation of our approach for numeric programs demonstrated results that are comparable to those of state-of-the-art tools.

As directions of future work we see the following. One direction is to develop a more precise path domain, similar to that of [MR05]. Having a domain that can represent, e.g., lasso-shaped paths would allow better handling of nested loops and maybe even extending our analysis to proving non-termination (rather than just finding existential recurrent sets). Another direction that will improve the analysis of numeric programs is to develop a specialized numeric refinement step. This specialized refinement will not exclude whole disjuncts from a candidate recurrent set, but will rather refine them (so that they satisfy the predicate CONT) improving the precision of the analysis. Finally, a possible direction is to adapt the approach to non-numeric abstract domains (e.g., to domains for shape analysis). To do so, we will need to replace lower widening with the appropriate domain specific extrapolation technique and to specialize the predicate CONT to support new kinds of statements.

3.A Constructing the Abstract Domain $\mathbb{D}_\sharp$

For $d, d_1, d_2 \in \mathbb{D}_\sharp$,

$$\bot_\sharp = \{l \mapsto \bot_{\mathrm{mp}} \mid l \in \mathbb{L}\}$$

$$\top_\sharp = \{l \mapsto \top_{\mathrm{mp}} \mid l \in \mathbb{L}\}$$

$$d_1 \sqsubseteq_\sharp d_2 \text{ iff } \forall l \in \mathbb{L}.\ d_1(l) \sqsubseteq_{\mathrm{mp}} d_2(l)$$

$$d_1 \sqcup_\sharp d_2 = \{l \mapsto d_1(l) \sqcup_{\mathrm{mp}} d_2(l) \mid l \in \mathbb{L}\}$$

$$d_1 \sqcap_\sharp d_2 = \{l \mapsto d_1(l) \sqcap_{\mathrm{mp}} d_2(l) \mid l \in \mathbb{L}\}$$

$$d_1 \triangledown_\sharp d_2 = \{l \mapsto d_1(l) \triangledown_{\mathrm{mp}} d_2(l) \mid l \in \mathbb{L}\}$$

$$d_1 \underline{\triangledown}_\sharp d_2 = \{l \mapsto d_1(l) \underline{\triangledown}_{\mathrm{mp}} d_2(l) \mid l \in \mathbb{L}\}$$

$$\mathrm{post}^\sharp(\mathbb{P}, d) = \{l \mapsto \bigsqcup\nolimits_{\mathrm{mp}} \{\mathrm{post}^{\mathrm{mp}}((l', l), d(l')) \mid (l', l) \in \mathbb{E}\} \mid l \in \mathbb{L}\}$$

$$\mathrm{pre}^\sharp(\mathbb{P}, d) = \{l \mapsto \bigsqcup\nolimits_{\mathrm{mp}} \{\mathrm{pre}^{\mathrm{mp}}((l, l'), d(l')) \mid (l, l') \in \mathbb{E}\} \mid l \in \mathbb{L}\}$$

3.B On Chaotic Iteration

Chaotic iteration is an important practical aspect, but for simplicity of presentation we do not take it into account in the definitions of the abstract computation, either in Section 2.3 where it is first introduced, or in the subsequent chapters (e.g., in (3.1) or (3.2) in this chapter). When the abstract domain is partitioned with program locations (like in this chapter, where $\mathbb{D}_\sharp = \mathbb{L} \to \mathbb{D}_{\mathrm{mp}}$), the simplistic formulation means that every step of the analysis will update (the mappings for) all the locations of an approximation in parallel. This is considered sub-optimal as in practice only a few locations will be updated in a single step. For example, in a forward analysis, like in (3.1), the first step will update the graph successors of the initial location, the second – the locations reachable in exactly two steps from the initial one and so on. In practice, it is common to update the locations on by one, picking them in some order, and this approach is called chaotic iteration.

We could amend the standard definition of the approximate computation to incorporate chaotic iteration. For the forward analysis of (3.1), this can be done in the following way. First, let $\mathrm{post}^\sharp_{\mathrm{iter}}$ be a stateful procedure (i.e., not a function in the mathematical sense), s.t. for $d \in \mathbb{D}_\sharp$

$$\mathrm{post}^\sharp_{\mathrm{iter}}(\mathbb{P}, d) = \{l_* \mapsto \bigsqcup\nolimits_{\mathrm{mp}} \{\mathrm{post}^{\mathrm{mp}}((l', l_*), d(l')) \mid (l', l_*) \in \mathbb{E}\}\} \cup \{d(l) \mid l \in \mathbb{L} \wedge l \neq l_*\}$$
$$\text{for some picked location } l_*$$

That is, $\mathrm{post}^\sharp_{\mathrm{iter}}$ every time picks a new location l_* and updates the mapping for it. Then, define the chain of approximants $\{f_i\}_{i \geq 0}$ as follows

$$f_0 = \{l_\vdash \mapsto \top_{\mathrm{mp}}; l \neq l_\vdash \mapsto \bot_{\mathrm{mp}}\}$$
$$f_i = f_{i-1} \triangledown_\sharp (f_{i-1} \sqcup_\sharp \mathrm{post}^\sharp_{\mathrm{iter}}(\mathbb{P}, f_{i-1})), \text{ for } i \geq 1$$

Finally, define the result of the analysis F to be the first f_j, s.t. $f_j = f_k$ for *all* $k \geq j$.

Now, the analysis requires two additional components. First component is an iteration strategy, i.e., the order in which the locations are updated. The strategy has to be fair, i.e., every location that can be updated should be updated eventually. Additionally, the strategy is expected to avoid making redundant updates. For example, in a sequential fragment program, we would expect the strategy to update the locations

in topological order.

Second component is a stopping condition, i.e., a way to detect that the chain of approximants has reached its stable limit and no update is possible in future. In many analyses, both components are implemented using a worklist. That is, the analysis maintains a worklist of locations that should be updated and in every iteration picks one (possibly, according to some priority). After updating a location (and if the corresponding value in the mapping was changed), the analysis adds its graph successor (for the case of forward analysis) to the worklist and proceeds to the next iteration. An example of a sophisticated worklist iteration algorithm is described in [ASV12]. In [Bou93b], François Bourdoncle offers a number of iteration strategies (and corresponding stopping conditions) that do not require a worklist.

For the abstract computation, the order in which the locations are updated does matter for both performance and precision of the result. If the strategy is fair, the result will be a sound approximation of the concrete fixed point, but different fair strategies may produce chains with different stable limits. For example, it is believed that for the nested loops (at least in numeric analyses) it is more optimal to update locations in the inner loop with higher priority that the locations in the outer loop.

Chapter 4

Finding Universal Recurrent Sets with Forward Analysis

In this chapter we present an algorithm that allows us to under-approximate universal recurrent sets of individual loops, but this time – in *structured* programs[1]. More specifically, we address another obstacle in computing recurrent sets via greatest fixed point characterizations, namely that backward analysis (analysis based on pre-condition or predecessor operations) is computationally expensive. This is less the case for numeric programs, but becomes an issue for heap-manipulating ones. Intuitively, this is because backward transformers exhibit non-determinism (e.g., *"what was the value of a pointer before it was updated?"*) in a way that is hard to deal with in shape analysis domains. For example, backward analysis with separation logic [Rey02] is known to be computationally harder than forward analysis [CYO01]. For shape analysis with 3-valued logic [SRW02], we are aware of a single attempt to approximate a fixed point of backward transformers [LA+07]. This analysis is much more complicated than forward analysis (e.g., the implementation is based on both TVLA [LMS04] – the original implementation of shape analysis with 3-valued logic – and the SPASS theorem prover [Wei+09]).

This motivated us to try and find a way to compute recurrent sets via forward analysis, so that the resulting procedure would be immediately applicable not only to numeric, but also to heap-manipulating programs. The main challenge of a forward approach is that to our knowledge there is no way to characterize recurrent sets in terms

[1] The reader is invited to re-visit Section 2.5 before proceeding.

of forward transformers[2]. Instead, we were able to produce a *condition* for a set of states in a structured program to be universally recurrent. The algorithm that we develop in this chapter will systematically explore the state space of individual loops in the program, searching for what we call a *recurrent component*, which is somewhat similar to the notion of an *end component* in a Markov decision process [BK08].

4.1 Background

We define the analysis for a *subset* of the language of structured programs.

If for a memory state $m \in \mathbb{M}$ and a statement $C \in \mathbb{C}$, there exists no memory state $m' \in \mathbb{M}$ s.t. $(m, m') \in T_{\mathbb{M}}(C)$, we say that the execution of C *diverges* from m. Under certain conditions, this definition agrees with the common one based on a small-step semantics: all execution postfixes starting from m are infinite, and there exists at least one. This is the case when both of the following holds:

(i) assumption statements appear only at the start of a branch or at the entry or exit of a loop (i.e., assumption statements cannot be used freely in the program):

$$C ::= a \mid C_1 \,;\, C_2 \mid ([\varphi] \,;\, C_1) + ([\psi] \,;\, C_2) \mid ([\psi] \,;\, C)^* \,;\, [\varphi]$$

(ii) branch and loop guard assumptions are exhaustive: $\varphi \vee \psi = 1$

Then, the only way for an execution to diverge (in the above sense) is to get stuck in an infinite loop. Therefore (as in Chapter 3) we are going to find recurrent sets of individual loops.

In the rest of the chapter we will focus on the loop statement:

$$C_{\text{loop}} = ([\psi_{\text{ent}}] \,;\, C_{\text{body}})^* \,;\, [\varphi_{\text{exit}}] \tag{4.1}$$

Here, C_{body} is the *loop body*; if ψ_{ent} holds, the execution may enter the loop body; if φ_{exit} holds, the execution may exit the loop; and $\psi_{\text{ent}} \vee \varphi_{\text{exit}} = 1$. What is important for

[2] We do not explore it in this work, but certain subsets of non-terminating behaviors can be characterized using forward transformers. More specifically, we can produce a characterization of states from which we can build an infinite sequence of predecessors. In particular this would include states that are visited infinitely often by a non-terminating execution, e.g., the set $x \in [50; 60]$ in Example 4.1. Cases when a set of states is not escaped, but no individual state in it is visited infinitely often, e.g., the set $x \in [100; +\infty)$ in Example 4.1, it seems, cannot be captured by such characterization.

```
1 while (x ≥ 1) {
2   if (x = 60) x ← 50;
3   x ← x + 1;
4   if (x = 100) x ← 0;
5 }
```

Figure 4.1: Program text for Example 4.1.

us is that this form of loop has a single point serving as both the entry and the exit. As currently formulated, our analysis relies on this property, although we anticipate that more complicated control flow graphs can be analyzed in a similar way.

Universal Recurrent Set of a Loop

For a loop as in (4.1), let us define a *projection of a universal recurrent set on the loop entry*. This is a set $R_\forall$, s.t.

$$R_\forall \subseteq [\![\neg\varphi_{\text{exit}}]\!]$$
$$\forall m \in R_\forall . \left(\forall m' \in \mathbb{M}. (m, m') \in T_{\mathbb{M}}(C_{\text{body}}) \Rightarrow m' \in R_\forall\right)$$

Intuitively, if an execution (of the corresponding unstructured program) reaches the loop entry in a memory state $m \in R_\forall$, it will stay in the loop forever. First, the execution cannot exit the loop from that state. Second, from Lemma 2.8, every terminating execution of the (graph of) loop body will lead to a memory state m' that also belongs to $R_\forall$. In the rest of the chapter, we will (somewhat ambiguously) call such set $R_\forall$ a *universal recurrent set of a loop*.

The conditions for a set of memory states to be universally recurrent are captured by Lemma 4.1 (in the concrete case) and Theorem 4.1 (in the abstract case).

Lemma 4.1. For a loop as in (4.1), the set $R \subseteq \mathbb{M}$ is universally recurrent (that is, R is a projection of a universal recurrent set on the loop entry) iff $\text{eval}(\neg\varphi_{\text{exit}}, R) = 1$ and $\text{post}(T_{\mathbb{M}}(C_{\text{body}}), R) \subseteq R$.

Proof. Follows from the definitions of eval, post, and universal recurrent set of a loop. □

Example 4.1. Consider the loop shown in pseudocode in Fig. 4.1 which will be our running example for this chapter.

Notice that if at the head of the loop x lies in the interval $[1;60]$ or x is greater or equal to 100, the loop will not terminate from that point. Indeed, if x starts between, 1 and 60, the loop will increment x until it reaches 60, then set it to 50, and the process will continue forever. If x starts at 100 or greater (and assuming that x is a mathematical integer), the loop will increment it indefinitely. Let us assume that $\mathbb{M} = (\mathbb{V} \to \mathbb{Z}) \cup \{\varepsilon\}$ (non-error memory states map program variables to integer values), and $\mathbb{V} = \{x\}$ (x is the only program variable). Then, we can say that the universal recurrent set of this loop is the following set of memory states:

$$R_\forall = \{x \mapsto n \mid (1 \leq n \leq 60) \vee (n \geq 100)\}$$

Now, let us look at the post-condition of $R_\forall$ w.r.t. the loop body (lines 2-4 of the pseudocode). If x starts between 1 and 59, it is only incremented at line 3 and not affected by lines 2 and 4, thus ending up between 2 and 60. If x is 60 it is set to 50 at line 2 and incremented to 51 at line 4. Finally, if x is greater or equal than 100, again, it is incremented at line 3 and not affected by lines 2 and 4, ending up greater or equal than 101. Thus,

$$\mathrm{post}(C_{\mathrm{body}}, R_\forall) = \{x \mapsto n \mid (2 \leq n \leq 60) \vee (n \geq 101)\} \subseteq R_\forall$$

4.1.1 Recurrent Sets in the Abstract

We are analysing structured programs (thus, the locations of a corresponding unstructured program are implicit) and the analysis will work in a memory abstract domain, where elements represent sets of memory states. Since the memory abstract domain is the main domain of the analysis in this chapter, we will denote it by $\mathbb{D}_\sharp$ instead of $\mathbb{D}_\mathfrak{m}$.

Now, let $\mathbb{D}_\sharp$ be *some* memory abstract domain, with least element $\bot_\sharp$, greatest element $\top_\sharp$, partial order $\sqsubseteq_\sharp$, and join $\sqcup_\sharp$ (later, we will introduce some structure into this domain). Every element, $A \in \mathbb{D}_\sharp$ represents a set of memory states $\gamma_\sharp(A) \subseteq \mathbb{M}$. Let the over-approximate versions of post and eval be given, s.t. for an *arbitrary* statement

$C \in \mathbb{C}$, an element $A \in \mathbb{D}_\sharp$, and a memory-state formula θ,

$$\gamma_\sharp(\text{post}^\sharp(C, A)) \supseteq \text{post}(T_\mathbb{M}(C), \gamma_\sharp(A))$$
$$\text{eval}^\sharp(\theta, A) \sqsupseteq_\mathscr{K} \text{eval}(\theta, \gamma_\sharp(A))$$

Theorem 4.1. For a loop as in (4.1), a memory abstract domain $\mathbb{D}_\sharp$, and an element $A \in \mathbb{D}_\sharp$, if $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, A) = 1$ and $\text{post}^\sharp(T_\mathbb{M}(C_{\text{body}}), A) \sqsubseteq_\sharp A$, then $\gamma_\sharp(A)$ is universally recurrent.

Proof. From the properties of $\text{eval}^\sharp$, $\text{eval}(\neg\varphi_{\text{exit}}, \gamma_\sharp(A)) = 1$. From the properties of $\text{post}^\sharp$ and $\gamma_\sharp$, $\text{post}(T_\mathbb{M}(C_{\text{body}}), \gamma_\sharp(A)) \subseteq \gamma_\sharp(\text{post}^\sharp(C_{\text{body}}, A)) \subseteq \gamma_\sharp(A)$. Then, universal recurrence of $\gamma_\sharp(A)$ follows from Lemma 4.1. □

Note that in Theorem 4.1, the post-condition is taken with respect to the loop body *without* the preceding assumption statement.

4.2 Finding a Universal Recurrent Set

In this chapter, we construct the memory abstract domain $\mathbb{D}_\sharp$ as a powerset domain. Let $\mathscr{L}_\mathfrak{m}$ be the underlying set of *abstract memory states*. $\mathscr{L}_\mathfrak{m}$ is partially ordered by $\sqsubseteq_\mathfrak{m}$, with least element $\bot_\mathfrak{m}$ and concretization $\gamma_\mathfrak{m} : \mathscr{L}_\mathfrak{m} \to \mathscr{P}(\mathbb{M})$. Elements of $\mathscr{L}_\mathfrak{m}$ are *abstract memory states*. For example, in a numeric analysis, $\mathscr{L}_\mathfrak{m}$ can be a set of polyhedra. In a shape analysis, $\mathscr{L}_\mathfrak{m}$ can be a set of individual 3-valued structures extended with an artificial bottom element. Then, elements of $\mathbb{D}_\sharp$ will be elements of $\mathscr{P}(\mathscr{L}_\mathfrak{m})$, i.e., *sets* of abstract memory states.

We want the partial order in $\mathbb{D}_\sharp$ to be the Hoare order. For $A_1, A_2 \in \mathbb{D}_\sharp$ (i.e., $A_1, A_2 \subseteq \mathscr{L}_\mathfrak{m}$), we define

$$A_1 \sqsubseteq_\sharp A_2 \text{ iff } \forall a_1 \in A_1.\ \exists a_2 \in A_2.\ a_1 \sqsubseteq_\mathfrak{m} a_2$$

We want concretization in $\mathbb{D}_\sharp$ to be pointwise. For $A \in \mathbb{D}_\sharp$, we define

$$\gamma_\sharp(A) = \bigcup \{\gamma_\mathfrak{m}(a) \mid a \in A\}$$

Most importantly, we want the over-approximate operations in $\mathbb{D}_\sharp$ to be pointwise. That is, for $A \in \mathbb{D}_\sharp$, arbitrary (atomic or compound) statement $C \in \mathbb{C}$, and a state for-

mula θ, we want the following to hold (this is *not* a definition)

$$\begin{aligned} \mathrm{post}^\sharp(C, A) &= \bigcup_{a \in A} \mathrm{post}^\sharp(C, \{a\}) \\ \mathrm{eval}^\sharp(\theta, A) &= \bigsqcup_{a \in A}{}_{\mathscr{K}}\, \mathrm{eval}^\sharp(\theta, \{a\}) \end{aligned} \tag{4.2}$$

We construct $\mathbb{D}_\sharp$ to be a powerset domain with pointwise operations for a reason. We will search for a recurrent set in the form of a *set* of abstract memory states (i.e., a subset of $\mathscr{L}_\mathfrak{m}$). This way, the recurrent set will be an element of $\mathbb{D}_\sharp$, and we will be able to claim soundness via Theorem 4.1. At the same time, basic steps of the analysis will work on individual abstract memory states (i.e., individual elements of $\mathscr{L}_\mathfrak{m}$). Pointwise operations ensure that element-based reasoning produces a sound set-based result.

We construct operations on $\mathbb{D}_\sharp$ from the underlying operations of $\mathscr{L}_\mathfrak{m}$ as follows.

If operations in $\mathscr{L}_\mathfrak{m}$ are given for individual elements In a numeric analysis, $\mathscr{L}_\mathfrak{m}$ may be the domain of intervals or polyhedra. In this case, we may be given over-approximate operations $\mathrm{post}^\mathfrak{m}$ and $\mathrm{eval}^\mathfrak{m}$ that work with individual elements of $\mathscr{L}_\mathfrak{m}$, i.e. $\mathrm{post}^\mathfrak{m} : \mathbb{C} \times \mathscr{L}_\mathfrak{m} \to \mathscr{L}_\mathfrak{m}$ and $\mathrm{eval}^\mathfrak{m} : \Theta \times \mathscr{L}_\mathfrak{m} \to \mathscr{K}$. In this case, for $A \in \mathbb{D}_\sharp$, an arbitrary (atomic or compound) statement $C \in \mathbb{C}$, and a state formula θ, we can define

$$\begin{aligned} \mathrm{post}^\sharp(C, A) &= \bigcup_{a \in A} \{\mathrm{post}^\mathfrak{m}(C, a)\} \\ \mathrm{eval}^\sharp(\theta, A) &= \bigsqcup_{a \in A}{}_{\mathscr{K}}\, \mathrm{eval}^\mathfrak{m}(\theta, a) \end{aligned}$$

Many examples in this chapter use interval domain. In this case, post-condition w.r.t. a given statement ($\mathrm{post}^\sharp$ for a given statement C) accepts a set of intervals as input and produces a set of intervals as output. For example, to compute $\mathrm{post}^\sharp(x \leftarrow x+1, \{[0;1], [2;+\infty)\})$, we will apply the increment transformer to every interval in the set separately, thus producing:

$$\mathrm{post}^\sharp(x \leftarrow x+1, \{[0;1], [2;+\infty)\}) = \{[1;2], [3;+\infty)\}$$

Similarly,

$$\mathrm{eval}^\sharp(x \geq 2, \{[0;1], [2;+\infty)\}) = \mathrm{eval}^\mathfrak{m}(x \geq 2, [0;1]) \sqcup_{\mathscr{K}} \mathrm{eval}^\mathfrak{m}(x \geq 2, [2;+\infty)) = 0 \sqcup_{\mathscr{K}} 1 = {}^1\!/_2$$

If operations in $\mathscr{L}_\mathfrak{m}$ are given for sets For some domains, we may be given operations that work on sets of elements of $\mathscr{L}_\mathfrak{m}$, i.e., $\text{post}^\mathfrak{m} : \mathbb{C} \times \mathscr{P}(\mathscr{L}_\mathfrak{m}) \to \mathscr{P}(\mathscr{L}_\mathfrak{m})$ and $\text{eval}^\mathfrak{m} : \Theta \times \mathscr{P}(\mathscr{L}_\mathfrak{m}) \to \mathscr{K}$. For example, this is the case for shape analysis with 3-valued logic. Then, for $A \in \mathbb{D}_\sharp$, arbitrary (atomic or compound) statement $C \in \mathbb{C}$, and a state formula θ, we can define[3]

$$\text{post}^\sharp(C, A) = \bigcup_{a \in A} \text{post}^\mathfrak{m}(C, \{a\})$$
$$\text{eval}^\sharp(\theta, A) = \bigsqcup_{a \in A, \mathscr{K}} \text{eval}^\mathfrak{m}(\theta, \{a\})$$

In either case, we require that $\text{eval}^\mathfrak{m}$ (as a consequence, $\text{eval}^\sharp$ will also be) is monotone: for a formula θ and $a_1, a_2 \in \mathscr{L}_\mathfrak{m}$, $a_1 \sqsubseteq a_2 \Rightarrow \text{eval}^\mathfrak{m}(\theta, a_1) \sqsubseteq_\mathscr{K} \text{eval}^\mathfrak{m}(\theta, a_2)$. Normally, $\text{eval}^\mathfrak{m}$ is given for atomic formulas, and for arbitrary formulas it is defined by induction over the formula structure, using 3-valued logical operators, possibly over-approximate with respect to $\sqsubseteq_\mathscr{K}$.

Also, we assume that the bottom element $\bot_\mathfrak{m}$, which represents unreachability, is transformed and evaluated precisely.

$$\gamma_\sharp(\{\bot_\mathfrak{m}\}) = \varnothing$$
$$\text{post}^\sharp(C, \{\bot_\mathfrak{m}\}) = \varnothing$$
$$\text{eval}^\sharp(\theta, \{\bot_\mathfrak{m}\}) = 1$$

In this chapter, we will want a way to split an abstract memory state (or a set of abstract memory states) into those that do satisfy some memory state formula θ and those that do not. For that we will use a post-condition w.r.t. an assumption statement – $\text{post}^\sharp([\theta], \cdot)$. We will abbreviate $\text{post}^\sharp([\theta], \cdot)$ as $[\theta, \cdot]^\sharp$.

We will also want a way to split an abstract memory state (or a set of abstract memory states) into erroneous and non-erroneous ones. For a set of abstract memory states $A \in \mathbb{D}_\sharp$, let $[\varepsilon, A]^\sharp$ be an operation that attempts to produce the smallest erroneous abstract memory state below A. Formally, we require that

$$[\varepsilon, A]^\sharp \sqsubseteq_\sharp A$$
$$\gamma_\sharp([\varepsilon, A]^\sharp) \supseteq \gamma_\sharp(A) \cap \{\varepsilon\}$$

[3] Even though $\text{post}^\mathfrak{m}$ and $\text{eval}^\mathfrak{m}$ operate on sets, they might not satisfy (4.2).

This is the same as taking a post-condition w.r.t. the statement [0] (assume false).

Similarly, let $[\neg\varepsilon, A]^\sharp$ be an operation that produces an over-approximation of non-error memory states of A:

$$[\neg\varepsilon, A]^\sharp \sqsubseteq_\sharp A$$
$$\gamma_\sharp([\neg\varepsilon, A]^\sharp) \supseteq \gamma_\sharp(A) \setminus \{\varepsilon\}$$

This is a non-standard operation that does not correspond to a post condition w.r.t. a statement, as statements are not allowed to recover from error.

4.2.1 Idea of the Algorithm

For a loop as in (4.1), if we find $X \in \mathbb{D}_\sharp$, s.t. $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, X) = 1$ and $\text{post}^\sharp(C_{\text{body}}, X) \sqsubseteq_\sharp X$, then $\gamma_\sharp(X)$ is *definitely* a recurrent set. The idea is to explore the state space of the program with forward analysis until such an X is found. We proceed as follows. Separately for every loop, we build a graph where vertices are elements of $\mathscr{L}_\mathfrak{m}$, all representing sets of concrete states at the loop head. We initialize the graph with some set of abstract memory states $I \in \mathbb{D}_\sharp$ and then repeatedly apply the transformer for the whole loop body, $\text{post}^\sharp(C_{\text{body}}, \cdot)$, to the vertices (treating them as singleton sets) and add the resulting elements to the graph as successors. Our experiments suggest that in many cases (when the program indeed has a universal recurrent set) a subset X of vertices satisfying the conditions of Theorem 4.1 will emerge as a result. To be able to efficiently find such a subset, we remember which elements are related w.r.t. abstract order $\sqsubseteq_\mathfrak{m}$, as a second kind of edges in the graph. Note that in case of nested loops, we analyze inner and outer loops separately; when analyzing the outer one, the effect of the inner needs to be summarized in an over-approximating way[4].

We use a number of heuristics to help the analysis. First, we try to distinguish states that take different paths through the loop body. In this work, we took a simplistic approach: we prefer to use a powerset domain where join is set union. This way, abstract memory states produced by different branches are not joined, i.e., $\text{post}^\sharp(C_1 + C_2, A) = \text{post}^\sharp(C_1, A) \sqcup_\sharp \text{post}^\sharp(C_2, A)$. In principle, a more involved trace partitioning [RM07]

[4]This means that we will need to treat the inner loop as a single statement and be able to compute an abstract post-condition w.r.t. this statement. More specifically, given the entry condition ψ'_{ent} and the body C'_{body}, we need to be able to compute $\text{post}^\mathfrak{m}((([\psi'_{\text{ent}}]\ ;\ C_{\text{body}})^*, a)$ for an abstract memory state a. This is done by approximating the limit of an ascending chain, as shown in (2.7), in Section 2.5.

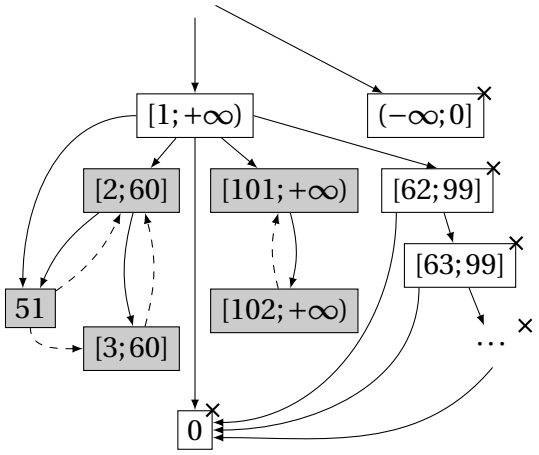


Figure 4.2: State graph for the program in Fig. 4.1.

could be introduced instead[5].

Second, we introduce additional case splits in the state graph, by applying assumption statements to elements. For a set of initial elements $I \in \mathbb{D}_\sharp$, we will actually initialize the graph with a set I', s.t.

$$I' = [\neg\psi_{\text{ent}}, I'']^\sharp \cup [\psi_{\text{ent}}, I'']^\sharp,$$
$$\text{where } I'' = [\varepsilon, I]^\sharp \cup [\neg\varepsilon, I]^\sharp$$

Before adding new elements to the graph, we will split them in a similar way. This encourages the algorithm to keep separately the erroneous and non-erroneous elements as well as those that may and may not enter the loop.

These heuristics are helpful when (as is often the case) there is a specific path through the loop body that infinite traces take. Then, the heuristics introduce control-flow distinctions and enable states taking such path to be partitioned from the others. But these heuristics may not be helpful when additional distinguishing power is needed for the *data* in states, e.g, when certain kinds of non-determinism are present, when non-termination depends on the properties of mathematical functions that the program implements, or when the abstract domain is not expressive enough to capture the states that take the interesting control paths.

Example 4.1 (continued)**.** Let us informally demonstrate how the algorithm that we propose works for the program in Fig. 4.1. Let us assume that x ranges over integers and

[5] As you may recall, trace partitioning is used in the analysis Chapter 3, but this analysis pre-dates it.

using intervals to represent its values. Since we do not know the initial value of x, we start with a graph consisting of a pair of elements: $\{(-\infty;0],[1;+\infty)\}$ – one represents the loop condition and another represents its complement. We then start adding new elements to the graph by computing $\text{post}^\sharp$ as described above, s.t. paths through the loop body are represented in a post-condition of an element by different disjuncts. For example, let us see what happens to $[1;+\infty)$ when it enters the loop. In line 2, we consider three cases. If $x < 60$, then the conditional body in line 2 is skipped, x is incremented at line 3, the conditional body in line 4 is skipped, and the output element is $[2;60]$. If $x = 60$, the conditional body in line 2 sets x to 50, at line 3 x is incremented, the conditional body in line 4 is skipped, and the output element is 51. If $x > 60$, the conditional body at line 2 is skipped and at line 3 x is incremented to $[62;+\infty)$. Then, if $x < 100$, the conditional body at line 4 is skipped, and the output element is $[62;99]$. If $x = 100$, the conditional body at line 4 sets x to 0, and the output element is 0. If $x > 100$, the conditional body at line 4 is skipped, and the output element is $[101;+\infty)$. Thus, $\text{post}^\sharp(C_{\text{body}},\{[1;+\infty)\}) = \{[3;60], 51, [62;99], 0, [101;+\infty)\}$. We add these elements to the graph and continue the exploration. Fig. 4.2 shows a state graph that will be produced this way after a number of steps. In the graph, boxes represent elements, and solid edges represent post-conditions. Note that in the graph, there exists a subset of elements $X = \{[2;60],[101;+\infty)\}$ has the desired property: $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, X) = 1$ and $\text{post}^\sharp(C_{\text{body}}, X) \sqsubseteq_\sharp X$, thus $\gamma_\sharp(X)$ is a recurrent set. In what follows, we discuss how to efficiently find such subset of elements if it exists. We revisit this example in Section 4.3.

For some domains (e.g., for shape analysis with 3-valued logic), the analysis benefits from case splits that $\text{post}^\sharp$ naturally performs. For example, when a program traverses a potentially cyclic list, $\text{post}^\sharp$ would likely consider a definitely cyclic list as a separate case. If the abstraction is expressive enough, the cyclic list case will appear as a separate vertex in the graph, and become part of a recurrent set.

Finally, the choice of the set of initial elements I matters. When the abstract domain is finite (and no widening is required) and the loop is not nested, we initialize the graph with a representation of states that reach the loop via the rest of the program, i.e., produced by the standard forward analysis of the preceding part of the program. In this case, the analysis will explore all the states reachable at the head of the loop, and the success relies only on how refined the resulting graph is. When the abstract domain is

infinite (e.g., for intervals or polyhedra) or for inner nested loops, we normally initialize the graph with a pre-fixpoint of $\text{post}^\sharp$. That is, we assume that initially, a standard forward analysis is run to produce a pre-fixpoint for every loop (an over-approximation of the memory states reachable at the loop entry). Starting with a set of elements below (w.r.t. $\sqsubseteq_\sharp$) a pre-fixpoint makes it less likely that the analysis terminates, as our procedure does not include widening. Starting with an element above a pre-fixpoint is more likely to drive the search towards the states unreachable from the program entry. Note that it is *sound* to start with any set of elements, and we sometimes start with just $\top_\sharp$.

Our procedure is sound (by Theorems 4.1 and 4.2), but incomplete: if we do not find a recurrent set after a number of steps, we do not know the reason: whether the loop does not have a universal recurrent set; or the abstraction and $\text{post}^\sharp$ are not expressive enough; or we did not explore enough states. Thus, for an infinite domain, the procedure might not terminate. So, we perform the exploration incrementally: we proceed breadth-first until some recurrent set is found. Then, we may decide to stop or to continue the search for a larger recurrent set.

4.2.2 Abstract Memory State Graph

For a loop as in (4.1), an *abstract memory state graph* (we will call it just *state graph*) is a graph $G = \langle V, E_p, E_c \rangle$, s.t.

- V is a *finite* non-empty set of vertices which are elements of $\mathscr{L}_\mathfrak{m}$: $V \subseteq \mathscr{L}_\mathfrak{m}$. Implicitly, these memory states belong to the loop entry location.
- There are two independent sets of edges: $E_c, E_p \subseteq V \times V$.
- E_p is a set of *post-edges*. For every element $a \in V$, *one* of the following holds:
  (i) there are no outgoing post-edges: $(\{a\} \times V) \cap E_p = \varnothing$ (this will mean that successors have not been explored by the analysis); or
  (ii) ψ_{ent} may hold in a; post-condition of a with respect to the loop body is not empty; the whole post-condition is in the graph; and it is connected to a by

post-edges:

$$\begin{aligned} & \mathrm{eval}^\sharp(\psi_{\mathrm{ent}}, \{a\}) \neq 0 \\ \wedge\ & \mathrm{post}^\sharp(C_{\mathrm{body}}, \{a\}) \neq \varnothing \\ \wedge\ & \mathrm{post}^\sharp(C_{\mathrm{body}}, \{a\}) \subseteq V \\ \wedge\ & (\{a\} \times V) \cap E_p = \{a\} \times \mathrm{post}^\sharp(C_{\mathrm{body}}, \{a\}) \end{aligned}$$

or

(iii) ψ_{ent} may hold in a; the post-condition of a is empty; a has $\bot_{\mathfrak{m}}$ as the only post-successor; and $\bot_{\mathfrak{m}}$ has a post-self-loop[6]:

$$\begin{aligned} & \mathrm{eval}^\sharp(\psi_{\mathrm{ent}}, \{a\}) \neq 0 \\ \wedge\ & \mathrm{post}^\sharp(C_{\mathrm{body}}, \{a\}) = \varnothing \\ \wedge\ & (\{a\} \times V) \cap E_p = \{(a, \bot_{\mathfrak{m}})\} \\ \wedge\ & (\bot_{\mathfrak{m}}, \bot_{\mathfrak{m}}) \in E_p \end{aligned}$$

- E_c is a set of *containment-edges*. For $a_1, a_2 \in V$, $(a_1, a_2) \in E_c$ iff $(a_1 \neq a_2 \wedge a_1 \sqsubseteq_{\mathfrak{m}} a_2)$. This definition forbids self-loops, and due to properties of $\sqsubseteq_{\mathfrak{m}}$, G cannot have containment cycles[7].

This is similar to the abstract reachability graphs built by modern model checking procedures (e.g., Impact [McM06] or analyses built within the CPAChecker framework [BHT07; BHT08]) and to the termination graphs built by AProVE [Gie+14]. Our algorithm, though, differs in the way we analyse the graph.

For a loop as in (4.1), a state graph $G = \langle V, E_p, E_c \rangle$, an element $a \in V$, and a set of elements $A \subseteq V$, let us define the successors in the graph as

$$\begin{aligned} \mathrm{post}^G(a) &= \{a' \in V \mid (a, a') \in E_p\} \\ \mathrm{post}^G(A) &= \{a' \in V \mid \exists a \in A.\ (a, a') \in E_p\} \end{aligned}$$

[6] This is added mostly for technical reasons. We want a way to distinguish the cases when the post-condition of an element have not been yet computed and when the post-condition of an element is the empty set. In the latter case, we make $\bot_{\mathfrak{m}}$ the post-successor of that element in the graph.

[7] In $\mathscr{L}_{\mathfrak{m}}$ as a partially ordered set, if $a_1 \sqsubseteq_{\mathfrak{m}} a_2 \sqsubseteq_{\mathfrak{m}} \cdots \sqsubseteq_{\mathfrak{m}} a_n \sqsubseteq_{\mathfrak{m}} a_1$ then $a_1 = a_2 = \cdots = a_n$ and thus all of them correspond to the same vertex in the graph. This fact, while mathematically obvious, has implications for implementation of the analysis: membership in the state graph should be based on semantic equivalence between elements, and it is not acceptable to use an approximate (e.g., some form of structural) equivalence that is inconsistent with the definition and implementation of $\sqsubseteq_{\mathfrak{m}}$.

For a loop as in (4.1) and a graph $G = \langle V, E_p, E_c \rangle$, a *recurrent component* is a set of elements $R \subseteq V$, s.t. every element $a \in R$, is non-error, cannot exit the loop, has at least one outgoing edge (post- or containment-):

$$\begin{aligned} &\mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, \{a\}) = 1 \\ \wedge\ &\exists a' \in V.\ (a, a') \in E_p \cup E_c \end{aligned}$$

and also *at least one* of the following is true:

(i) a has a containment-edge into R: $\exists a' \in R.\ (a, a') \in E_c$; or

(ii) the outgoing post-edges of a lead exclusively into R: $\mathrm{post}^G(a) \neq \varnothing \wedge \mathrm{post}^G(a) \subseteq R$.

Example 4.1 (continued)**.** The entity in Fig. 4.1 represents a state graph, with some modifications made for clarity. Boxes represent the vertices of the graph, which are members of an interval domain. Solid arrows represent post-edges. Dashed errors represent containment edges, but for clarity, *not all containment edges are shown* (for example, there should be a number containment edged going into the vertex $[1; +\infty)$). Notice that vertices with grey background form a recurrent component (actually, containment edges are only shown between the vertices in the component).

Lemma 4.2. The union of two recurrent components is a recurrent component.

Lemma 4.3. In a state graph G, there exists a unique maximal (possibly, empty) recurrent component.

Proof. Lemma 4.2 follows from the definition of recurrent component. Lemma 4.3 follows from Lemma 4.2 and finiteness of G. □

Theorem 4.2. For a loop as in (4.1) and a state graph $G = \langle V, E_p, E_c \rangle$ we say $X \subseteq V$ is *fully closed* if $\mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, X) = 1$, $\forall a \in X.\ \mathrm{post}^G(a) \neq \varnothing$, and $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) \sqsubseteq X$. (Note that in this case, $\gamma_\sharp(X)$ is a recurrent set.)

Then, for every state graph G:

(i) For a recurrent component R, there exists a fully closed $X \subseteq R$ s.t. $\gamma_\sharp(X) = \gamma_\sharp(R)$.

(ii) For a fully closed X, there exists a recurrent component $R \supseteq X$, s.t. $\gamma_\sharp(R) = \gamma_\sharp(X)$.

Proof. We present the proof in Appendix 4.A □

4.2.3 The Algorithm

The algorithm, whose main body is shown in pseudocode in Figures 4.3 and 4.4, is applied individually to every loop in a program. Initially, we call the procedure *FindFirst* giving it the set of elements $I \subseteq \mathscr{L}_{\mathfrak{m}}$ to start the search from (normally, a loop invariant). After performing initialization, *FindFirst* calls *FindNext* once. *FindNext* contains a loop in which we build the state graph $G = \langle V, E_p, E_c \rangle$. In every iteration, proceeding in breadth-first order, we pick from the worklist F an element without post-edges and add its successors to the graph, together with relevant post- and containment-edges. This happens in lines 3–10 of Fig. 4.4; new elements and post-edges are created by *MakeNewElements* shown in Fig. 4.5. We choose not to explore the successors of an element belonging to a recurrent component (lines 4–6) even though when $\text{post}^\sharp$ is non-monotone, they might lie outside the recurrent component. Similarly, we do not explore the successors of a must-exiting element, even if ψ_{ent} may hold in it. If adding new elements and edges could create a larger recurrent component, we call *FindRecComp* to search for it (lines 13–15). If a new recurrent component is found, we return 1, and *Rec* contains those elements of the component found so far that have no outgoing containment-edges (lines 16–26). If we wish to find a larger recurrent component, we can call *FindNext* again to resume the search. If the search terminates and no new recurrent component can be found, the procedure returns 0.

For every abstract element $a \in V$, we maintain the *status* as follows.

We say that an element $a \in V$ *must exit*, $mustE(a) = 1$, if all executions starting in it exit the loop or reach an error memory state, i.e., if it is definitely the case that for every concrete memory state $m \in \gamma_\sharp(\{a\})$ the loop eventually terminates or reaches an error. We mark a as must-exiting if

(i) $\text{eval}^\sharp(\psi_{\text{ent}}, \{a\}) = 0$ (note that this is also the case for erroneous abstract memory states); or if

(ii) all post-successors of a are already must-exiting; or if

(iii) there exists a larger (w.r.t. $\sqsubseteq_{\mathfrak{m}}$) element that is already must-exiting.

We say that an element $a \in V$ *may exit*, $mayE(a) = 1$, if we know that it cannot be part of a recurrent component. We mark a as may-exiting if

Procedure: Find First
Input: Set of initial elements $I \subseteq \mathscr{L}_\mathfrak{m}$
Output: Whether recurrent set *Rec* was updated
Global variables: State graph $G = \langle V, E_p, E_c \rangle$, worklist $F \subseteq \mathscr{L}_\mathfrak{m}$,
current recurrent set $Rec \subseteq \mathscr{L}_\mathfrak{m}$

```
1 procedure FindFirst(I):
2   for A ∈ L_m do
3     mayE(a) ← mustE(a) ← rec(a) ← 0
4     unk(a) ← 1
5   endfor
6   MakeNewElements(I, nil)
7   F ← {a ∈ V | ¬mustE(a) ∧ a ≠ ⊥_m}
8   FindNext()
9 endproc
```

Figure 4.3: Procedure *FindFirst* that finds the first recurrent set.

(i) it is must-exiting or if $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, \{a\}) \neq 1$; or if

(ii) $\text{post}^\sharp$ is monotone and a has a post-successor that is already may-exiting; or if

(iii) $\text{post}^\sharp$ is monotone, and there exists a smaller (w.r.t. $\sqsubseteq_\mathfrak{m}$) already may-exiting element.

We say that an element $a \in V$ is *recurrent*, $rec(a) = 1$, if it is a part of a recurrent component. If $\text{post}^\sharp$ is monotone, we also mark as recurrent all successors of a recurrent element. Here, the term *recurrent* is overloaded. For a recurrent *element* $a \in V$, $\gamma_\sharp(\{a\})$ is in general not a recurrent *set* itself, but is included in some recurrent set.

Otherwise, the element $a \in V$ is *unknown*, $unk(a) = 1$, i.e., $unk(a) \Rightarrow (\neg mayE(a) \wedge \neg rec(a))$. This is the case if $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, \{a\}) = 1$, and the element may become a part of a recurrent component, but it is not part of the recurrent component found so far.

Lemma 4.4. May-exiting elements cannot be part of a recurrent component.

Proof. We give the proof in Appendix 4.A. □

When searching for a recurrent component, it is only necessary to consider unknown and recurrent elements, therefore every step of the algorithm only creates new

Procedure: Find Next
Output: Whether recurrent set *Rec* was updated
Global variables: State graph $G = \langle V, E_p, E_c \rangle$, worklist $F \subseteq \mathscr{L}_{\mathfrak{m}}$,
current recurrent set $Rec \subseteq \mathscr{L}_{\mathfrak{m}}$

```
1  procedure FindNext():
2    while F ≠ ∅ do
3      a ← first(F); F ← F \ {a}
4      if mustE(a) ∨ rec(a) then
5        continue
6      endif
7      newPost ← MakeNewElements(post♯(C_body, {a}), a)
8      E_c^+ ← {(a', a'') ∈ V × V | unk(a') ∧ (unk(a'') ∨ rec(a'')) ∧ a' ⊑ a'' ∧
                                    (a'' ∈ newPost ∨ a' ∈ newPost)}
9      E_c ← E_c ∪ E_c^+
10     F ← F ∪ {a' ∈ newPost | ¬mustE(a') ∧ a' ≠ ⊥_m}
11     PropagateStatus()
12     R ← ∅
13     if (newPost = ∅ ∧ (∀a' ∈ post^G(a). unk(a') ∨ rec(a'))) ∨ E_c^+ ≠ ∅ then
14       R ← FindRecComp()
15     endif
16     if R ≠ ∅ then
17       for a ∈ R do
18         rec(a) ← 1; unk(a) ← 0
19       endfor
20       PropagateStatus()
21       Rec' ← Rec
22       Rec ← {a' ∈ V | rec(a') ∧ ({(a', a'') | a'' ∈ V ∧ rec(a'')} ∩ E_c = ∅)}
23       if (Rec ≠ Rec') then
24         return 1
25       endif
26     endif
27   endwhile
28   return 0
29 endproc
```

Figure 4.4: Procedure *FindNext* that finds the next recurrent set.

containment-edges between unknown elements or from an unknown to a recurrent element.

Note that when new elements or edges are added to the graph, or the status of an existing element changes, we make a call to *PropagateStatus*. *PropagateStatus* propagates the statuses through the edges of the graph according to the following rules. For an element a:

(i) if $\text{post}^G(a) \neq \varnothing \wedge \forall a' \in \text{post}^G(a).\ mustE(a')$, then $mustE(a)$

(ii) if $mustE(a)$, then $\forall a'.\ (a', a) \in E_c \Rightarrow mustE(a')$

(iii) if $\text{post}^G(a) \neq \varnothing \wedge \forall a' \in \text{post}^G(a).\ rec(a')$, then $rec(a)$

(iv) if $rec(a)$, then $\forall a'.\ (a', a) \in E_c \Rightarrow rec(l')$

Additionally, if $\text{post}^\sharp$ is monotone[8]:

(v) if $\exists a' \in \text{post}^G(a).\ mayE(a')$, then $mayE(a)$

(vi) if $mayE(a)$, then $\forall a'.\ (a, a') \in E_c \Rightarrow mayE(a')$

(vii) if $rec(a)$, then $\forall a' \in \text{post}^G(a).\ rec(a')$

(viii) if $mustE(a)$, then $\forall a' \in \text{post}^G(a).\ mustE(l')$

Rules (i) and (ii) are derived from the definition of must-exiting element. Rules (iii) and (iv) mark as recurrent those elements that would anyway be included in a recurrent component next time *FindRecComp* is called. Rules (v) and (vi) are derived from the definition of may-exiting elements. Rule (vii) is for the case when for some a, first its post-condition is computed, and later, a is marked as recurrent by rule (iv). If $\text{post}^\sharp$ is monotone, the successors of a would eventually become part of a recurrent component. Similarly, rule (viii) is for the case when for some a, first its post-condition is computed, and later, a is marked as must-exiting by rule (ii). If $\text{post}^\sharp$ is monotone, the successors of a would eventually be marked as must-exiting. This all is not necessary for the correctness: every element that *PropagateStatus* marks as may- or must-exiting,

[8] As you can see, monotonicity of $\text{post}^\sharp$ (which would follow if post^m is monotone) is helpful for the analysis. Usually, if the analysed loop is non-nested (i.e., its body is a loop-free program), the abstract post-condition w.r.t. its body will be monotone. In case of nested loops, *non*-monotonicity can be introduced by widening, which will be required in a numeric analysis to over-approximate the effect of an inner loop when analysing the outer one.

cannot be part of a recurrent component, and every element that it marks as recurrent would eventually become a part of a recurrent component anyway. But this allows to eliminate unknown elements earlier, create fewer containment-edges, and search for recurrent component in a smaller portion of the graph.

Fig. 4.5 shows the procedure *MakeNewElements* that adds new elements to the graph. Given a set of abstract elements $A \subseteq \mathscr{L}_{\mathfrak{m}}$ and a predecessor element $a_p \in V$, it adds abstract elements corresponding to A to the graph and creates post-edges from a_p to them. Every $a \in A$ is split into a number of elements with the assumption transformer, then is possibly marked as may- or must- exiting depending on the values of φ_{exit} and ψ_{ent} and added to the graph together with a post-edge from a_p. The procedure returns the set N of new elements produced from A that were not present in the graph before.

Fig. 4.6 shows the procedure *FindRecComp* that finds a (subset of a) recurrent component among the unknown elements. We call it from the procedure *FindNext* when a new containment-edge is created or an element is discovered such that all its outgoing post-edges lead to existing unknown or recurrent elements (i.e., when a larger recurrent component could emerge). It starts the search with the whole set of unknowns as the candidate C and iteratively removes the elements C^{-} that make the candidate violate the definition of recurrent component. Observe that *FindRecComp* works incrementally: assuming that R is a set of elements that are currently marked as recurrent (i.e., R is the recurrent component found so far), the procedure produces the largest set C, s.t. $C \cup R$ is a recurrent component. In general, C itself might not be a recurrent component.

Theorem 4.3. For an abstract state graph $G = \langle V, E_p, E_c \rangle$ and some recurrent component $R \subseteq V$, *FindRecComp* produces $C \subseteq V$ such that $C \cup R$ is the maximal recurrent component of G.

Proof. We give the proof in Appendix 4.A. □

4.3 Examples

We now demonstrate how our analysis can be successfully applied to numeric and heap-manipulating programs. Examples 4.1 and 4.2 present **Numeric Programs**. Pro-

Procedure: Make New Elements
Input: Set of elements $A \subseteq \mathscr{L}_{\mathfrak{m}}$ and a predecessor element $a_p \in \mathscr{L}_{\mathfrak{m}}$
Output: Set of new elements $N \subseteq \mathscr{L}_{\mathfrak{m}}$
Global variables: State graph $G = \langle V, E_p, E_c \rangle$

```
1  procedure MakeNewElements(A, a_p):
2    N ← ∅
3    if A = ∅ then
4      A' ← {⊥_m}
5    else
6      A' ← [ε, A]^♯ ∪ [¬ε, A]^♯
7      A' ← [ψ_ent, A']^♯ ∪ [¬ψ_ent, A']^♯
8    endif
9    for a ∈ A' do
10     if l_p ≠ nil then
11       E_p ← E_p ∪ (a_p, a)
12     endif
13     if a ∉ V:
14       if eval^♯(ψ_ent, {a}) = 0 then
15         unk(a) ← 0
16         mayE(a) ← mustE(a) ← 1
17       elseif eval^♯(¬φ_exit, {a}) ≠ 1 then
18         unk(a) ← 0
19         mayE(a) ← 1
20       endif
21       V ← V ∪ a
22       N ← N ∪ a
23       if a = ⊥_m then
24         E_p ← E_p ∪ {(a, a)}
25       endif
26     endif
27   endfor
28   return N
29 endproc
```

Figure 4.5: Procedure *MakeNewElements* that adds new elements to the graph. New elements are unknown unless marked otherwise.

Procedure: Find Recurrent Component
Output: Extension of the recurrent component $C \subseteq V$
Global variables: State graph $G = \langle V, E_p, E_c \rangle$

```
1  procedure FindRecComp() :
2    C ← {a ∈ V | unk(a)}
3    R ← {a ∈ V | rec(a)}
4    while 1 do
5      C⁻ ← {a ∈ C | {(a, a') | a' ∈ C ∪ R} ∩ E_c = ∅ ∧
                        (post^G(a) = ∅ ∨ post^G(a) ⊈ C ∪ R)}
6      if C⁻ = ∅ then break endif
7      C ← C \ C⁻
8    endwhile
9    return C
10 endproc
```

Figure 4.6: Finding a recurrent component.

gram variables range over integers, and we use intervals to represent their values. (Examples 4.3, 4.4, and 4.5 will present heap-manipulating programs.)

Example 4.1 (continued)**.** Let us revisit the program in Fig. 4.1 and its state graph Fig. 4.2. The graph is shown at a stage when the algorithm cannot find a larger recurrent component, and *FindNext* returns 0. The recurrent component is shown greyed, post-edges are solid, containment edges are dotted, and for clarity, containment-edges to and from may-exiting elements are not displayed. The element $[1; +\infty)$ is may-exiting, and must-exiting elements are marked with a cross. As a result, we find recurrent set $\{[2;60], [101; +\infty)\}$. Note that the states $x = 1$ and $x = 100$ are lost compared to the maximal recurrent set, and the discovered recurrent set is closed under application of the forward transformer, but not the backward transformer. This can be the case for some other tools based on forward semantics. For example, the tool E-HSF [BPR13] when presented with this example, may report the recurrent set to be $\{[4;60], [100; +\infty)\}$. Also, note the set of must-exiting elements (on the right side of the graph). While our algorithm often succeeds in proving that a recurrent set exists, it behaves badly when no recurrent set can be found. For example, in this case, it had to enumerate all elements of the form $[62;99], [63;99], [64;99]$, and so on. Finally, note that our procedure

```
1  days ← a number ≥ 0
2  year ← 1980
3  while (days > 365) {
4    if (leap(year)) {
5      if (days > 366) {
6        days ← days − 366;
7        year ← year + 1;
8      }
9    } else {
10     days ← days − 365;
11     year ← year + 1;
12   }
13 }
```

Figure 4.7: Demonstration of a real-life non-termination bug.

did terminate, although the abstract domain is infinite and we did not take measures to guarantee termination.

Example 4.2. Let us now revisit the program in Fig. 4.7 (which already appeared in Example 1.1). We presented this program to an implementation of our algorithm (with some splitting heuristics for modulo operation) with the starting element being the loop invariant: $year \geq 1980 \wedge days \geq 0$. Every call to *FindNext* extends the recurrent set with a single element: $year = 1980 \wedge days = 366$, $year = 1984 \wedge days = 366$, $year = 1988 \wedge days = 366$, and so on. The abstract domain was not expressive enough to infer that every leap year causes non-termination. Also, because the analysis is forward-only, it did not explore the predecessors of those elements: e.g., from the state $year = 1983 \wedge days = 731$, the loop also diverges, but this was not discovered by the tool. Still, we count this result as success: our approach does expose the bug even if it does not find all inputs for which the bug manifests.

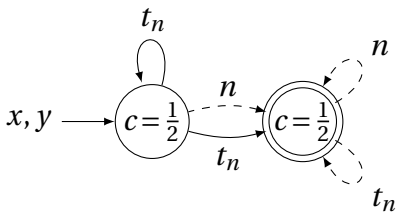


Figure 4.8: Acyclic list with 2+ elements.

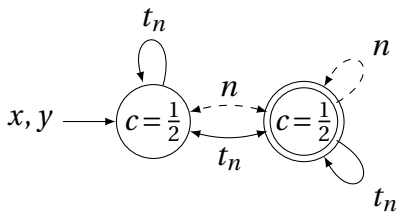


Figure 4.9: Cyclic list with 2+ elements.

```
1 | y ← x;
2 | while (y ≠ null ∧ ¬c(y))
3 |     y ← (y → n);
```

Figure 4.10: Linear search in a non-cyclic list.

```
1 | y ← x;
2 | while (y ≠ null)
3 |     y ← new record;
4 |     if (y ≠ null) {
5 |         (y → n) = x;
6 |         x ← y;
7 |     }
```

Figure 4.11: Prepending to a non-empty list.

Shape Analysis Examples

Examples 4.3, 4.4, and 4.5 present heap-manipulating programs. We use 3-valued logic [SRW02] to represent heaps, and build the analysis on top of the tool TVLA[9] [LMS04].

In Appendix 2.B, we gave a brief description of shape analysis with 3-valued logic that is sufficient to understand the examples. For more information on shape analysis with 3-valued logic, please refer to Sagiv et al. [SRW02] and related papers [RSL10; Arn+06; LMS04].

Example 4.3. One source of non-termination in heap-manipulating programs is incorrect traversal of cyclic data structures. Fig. 4.10 shows a procedure that searches a list pointed to by x for an element y s.t. the condition $c(y)$ holds. The search terminates when such y is found or when the end of the list is reached, and it does not handle cyclic lists correctly. In this and the next example, the initial statement: $y \leftarrow x$ – is disregarded by the analysis and only emphasizes for the reader that when the loop is reached for the first time, both x and y point to the head of the list. Due to canonical abstraction[10], the set of 3-valued structures that we can explore is finite, and there is no

[9] `http://www.cs.tau.ac.il/~tvla/`, last accessed in May 2016.

[10] In shape analysis with 3-valued logic, abstract transformers are usually designed in a way that they

need to perform pre-analysis for the loop invariant. Thus, we analyze the loop starting with the set of elements containing cyclic and acyclic lists with both x and y pointing to the head and with unknown value of c for all the cells: the structures shown in Figures 4.8 and 4.9, plus structures to represent single-element lists and an empty list. As our tool proceeds, it reports as the recurrent set all the heaps that cause non-termination of the loop, i.e., in this case – the cyclic lists where the condition c is false for all the elements. One of such lists (with three or more elements, y pointing into the list) is shown in Fig. 4.12.

Example 4.4. Another interesting class of bugs in heap-manipulating programs is related to heap allocation. Sometimes, models of programs do not take into account that heap allocation can fail. For example, in a real program, an infinite loop performing allocation would usually lead to an out-of-memory error and may consume much time and system resources. But in a model of the program this may appear as potential non-termination. Fig. 4.11 shows a program that repeatedly prepends a newly allocated element to a (non-empty) list. The loop is supposed to terminate if the allocation fails, but this is not possible in our TVLA model. The state graph for the example is shown in Fig. 4.13. The initial elements are: a list with two or more elements (element 1, as shown in Fig. 4.8), an empty heap (2), and a single-element list (3). The empty heap is must-exiting, and the elements 1, 3, and 4 (list with exactly two elements) form the recurrent set. Element 4 does not have an outgoing post-edge as the algorithm finishes before the post-condition of the element is computed. Note the post-loop on element 1. Because of canonical abstraction, the post-condition of a list with two or more elements is again a list with two or more elements. On one hand, the analysis loses track of the length of the list. On the other hand, abstracting from the length of the list allows us to produce a compact summary of the recurrent set.

Example 4.5. Fig. 4.14 shows a fragment of a device driver procedure that has a non-termination-related bug discovered by a termination prover [Ber+06]. The fragment is a loop that traverses a cyclic doubly-linked list. Every `entry` of the list is embedded in

produce structures belonging to a certain *finite* subset of 3-valued structures. One possible finite subset consists of what is called *bounded structures*. The analysis fixes a set of unary *abstraction predicates* (usually – just all unary predicates). Then, bounded structures are those where all nodes have distinct values of abstraction predicates. Finiteness of the image of the transformers allows to claim termination of analyses.

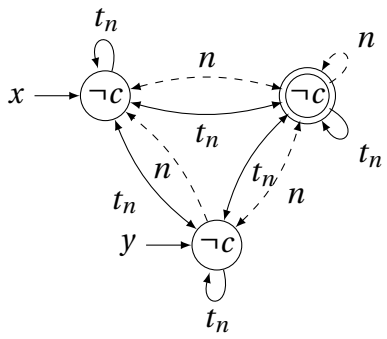


Figure 4.12: Example of a cyclic list where c is false for all elements.

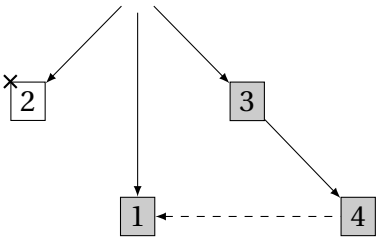


Figure 4.13: State graph for the program in Fig. 4.11. Element 1 is shown in Fig. 4.8. Grayed are recurrent elements and must-exiting element is marked with a cross.

a larger record and line 4 extracts a pointer to the containing record. If the condition in line 7 holds, the current `entry` is removed from the list in line 8. If the condition in line 9 also holds, the loop terminates. Otherwise, in line 12 the current entry's next-pointer (`Flink`) gets directed to the entry itself and the loop continues. In the latter case, the execution will stay in the loop as long as the conditions in lines 7 and 9 continue to hold for the entry.

Fig. 4.15 shows a simplified version of the loop that can be presented to our tool. Variable x always points to the head of the list, and y traverses the list. We did not have a template for working with cyclic doubly linked lists in TVLA, and we use a singly linked list, and explicitly track the pointer y_p to the previous list element. We also unify the list entries and containing records. To handle the early return in line 10, we manually transform the program to use an auxiliary condition $c(x)$.

What happens to conditions in lines 7 and 9 is more important. We need to abstract them, as they need much context to be modelled precisely. One option is to abstract them as non-deterministic conditions, but in this case there will be no universal non-termination, as there will always be the possibility for the execution to reach the return statement in line 10. But it may also be sound to model these conditions as predicates on list entries, s.t. they are not changed in the loop body, and initially we do not know whether they hold. In Fig. 4.15 we assume that this is the case, and introduce a pair of predicates k_1 and k_2 for the conditions in line 7 and 9 respectively. With the above assumptions, we were able to find a recurrent set of the program fragment. Fig. 4.16 shows an example of a 3-valued structure belonging to the discovered recurrent com-

ponent[11]. Notice the node pointed to by y where k_1 definitely holds, and k_2 definitely does not. Due to a logical error in the program, this node became detached from the original list, and the traversal along its self-loop never terminates.

This example shows that our approach can in principle handle real-world non-termination problems. At the same time, it points out some limitations. First, universal recurrent sets are fragile. Even though a non-termination bug may cause the program to have one (this is the case in Example 4.2 and may be the case in this example), it may be hard to build an abstraction that preserves it and does not introduce spurious terminating traces from every interesting state. In this example, building such abstraction requires certain knowledge about the context.

Second, the recurrent sets that we discover are genuine, but may not be reachable from the program entry. This can become an inconvenience when this procedure is used for debugging. For example, in this case we could not identify the input states that reach the recurrent set. In particular, just from the results of the analysis, we cannot see whether there exists a *valid* input that causes non-termiantion, or whether non-termination is caused by passing as an argument some misformed data structure.

4.4 Related Work

The problem of finding a universal recurrent set seems to be somehow less general than the problem of finding an existential recurrent set[12]. If a program has a reachable universal recurrent set, this means that it has (under certain conditions, recall Section 4.1) a non-terminating execution, but the inverse is not true. Due to non-determinism (e.g., see Example 3.1), a program may have non-terminating executions, but no universal recurrent set (when every state in a non-terminating execution, we can build a terminating execution postfix). The analysis of non-deterministic programs is important (it can be used to model the environment, to abstract away from details, etc) and probably this is why finding universal recurrent sets is a less popular

[11] For this program, the analysis (as suggested by the authors of TVLA) actually tracks a predicate stating whether or not a cell lies on a cycle. Also, it does not track binary reachability between cells, only unary reachability from variables. For clarity, we do not show the evaluation of the cyclicity and unary reachability predicates in the picture.

[12] One could also argue that it is a simpler one. Existence of a universal recurrent set is a problem complementary to reachability of exit location and thus is a safety property. Existence of an existential recurrent set is a liveness property.

```
1  for (entry = DeviceExtension->ReadQueue.Flink;
2       entry != &DeviceExtension->ReadQueue;
3       entry = entry->Flink) {
4    irp = (IRP *)((CHAR *)(entry)-(ULONG *)
5      (&((IRP *)0)->Tail.Overlay.ListEntry));
6    stack = IoGetCurrentIrpStackLocation (irp);
7    if (stack->FileObject == FileObject) {
8      RemoveEntryList(entry);
9      if (IoSetCancelRoutine (irp, NULL)) {
10       return irp;
11     } else {
12       InitializeListHead (&irp->Tail.Overlay.ListEntry);
13     }
14   }
15 }
```

Figure 4.14: Program fragment exhibiting a non-termination bug when manipulating a cyclic list.

problem. Still, it has seen some attention.

Cook et al. [Coo+14] analyze *linear* over-approximations of programs and then use Farkas' lemma to find universal recurrent sets. Their soundness result is similar to ours and is more general: they state it for arbitrary transition systems and require a property of *upward termination* (for every concrete final state, the corresponding abstract state is also final) which for us implicitly holds. Note that linear abstractions have not yet demonstrated to be very effective for analyzing heap-manipulating programs.

Larraz et al. [Lar+14] use the notion of an edge-closed quasi-invariant (a set of states that, once reached, cannot be escaped) as a generalization of recurrent set. They encode the search for such set as a max-SMT problem.

Le et al. propose a specification logic and an inference algorithm [LQC15] that can capture the absence of terminating behaviors.

Velroyen and Rümmer developed one of the early non-termination analysis [VR08]. They propose a template and a refinement scheme to infer invariants proving that terminating states of a program are unreachable.

```
1  y ← x;
2  y_p ← y;
3  y ← (y → n);
4  while (x ≠ y ∧ ¬c(x)) {
5     if (k_1(y)) {
6        (y_p → n) ← (y → n);
7        if (k_2(y))
8           c(x) ← 1;
9        else
10          (y → n) ← y;
11    }
12    if (¬c(x)) {
13       y_p ← y;
14       y ← (y → n);
15    }
16 }
```

Figure 4.15: Simplified version of the program in Fig. 4.14.

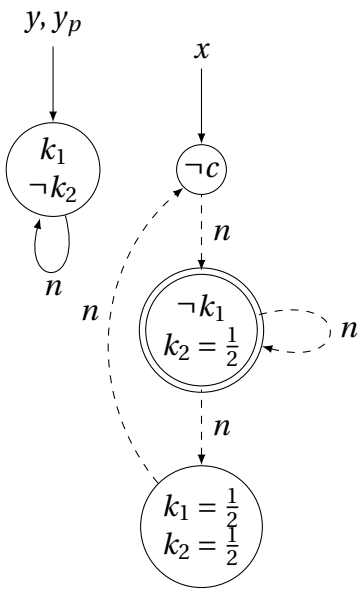


Figure 4.16: Sample structure from a recurrent component in Example 4.5.

4.5 Chapter Conclusion and Future Work

In this chapter, we described a forward technique for finding recurrent sets in imperative programs, where loops of a specific form are the source of non-termination. The recurrent sets that we produce are genuine, but may not be reachable from the program entry. We applied our analysis to numeric and heap-manipulating programs and were successful if (i) we were able to capture the paths through the program that infinite traces take, and (ii) we were able to perform enough case splits to isolate the recurrent set into a separate set of elements. The latter point can benefit from heuristics in some cases.

Our analysis only admits structured programs without `goto` statements, with restricted form of loops: `while`-loops without statements that affect control flow (`break`, `continue`, etc). One direction for future work is to enable the analysis of a larger class of loops: either by introducing relevant program transformations and studying their effect on the outcome of the analysis or by extending the technique to handle more complicated control flow graphs.

Another direction is to solidify the analysis: eliminate the need for a separate forward pre-analysis by weaving it into the main algorithm, introduce a proper trace partitioning, etc. Alternatively, we could try to re-formulate the analysis within the framework of another existing model checking procedure, e.g., Impact, as the way we construct the abstract state graph is already similar to what modern model checkers do. There exist extensible tools, like CPAChecker, that facilitate this kind of integration of different analyses.

Note that the analysis of this chapter will be able to find a recurrent set only if it is materialized as a set of elements in a state graph. This does not always happen naturally. For example, recall the program in Fig. 3.5. The loop does have a universal recurrent set, but non-termination relies on making a specific choice of y before the loop. Applying the analysis of this chapter directly will not materialize a recurrent set in the state graph, but we anticipate that there exist *heuristics* (possibly, including those based on widening techniques) that will partition the results of post-condition operation in a certain way and allow to materialize a recurrent set. Thus, developing such heuristics is another possible research direction.

4.A Omitted Proofs

Theorem 4.2. For a loop as in (4.1) and a state graph $G = \langle V, E_p, E_c \rangle$ we say $X \subseteq V$ is *fully closed* if $\mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, X) = 1$, $\forall a \in X.\ \mathrm{post}^G(a) \neq \varnothing$, and $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) \sqsubseteq X$. (Note that in this case, $\gamma_\sharp(X)$ is a recurrent set.)

Then, for every state graph G:

(i) For a recurrent component R, there exists a fully closed $X \subseteq R$ s.t. $\gamma_\sharp(X) = \gamma_\sharp(R)$.

(ii) For a fully closed X, there exists a recurrent component $R \supseteq X$, s.t. $\gamma_\sharp(R) = \gamma_\sharp(X)$.

Proof. **(i)** Let R be a recurrent component. Let X be the set of elements in R without outgoing containment-edges into R. This set is always not empty. Since R is finite, if every element in R had a containment-edge into R, there would be an infinite containment path within R, and therefore – a containment cycle, which is assumed never to happen. Since elements in X have no containment-edges into R, it must be that every element in X has an non-empty set of outgoing post-edges, all leading into R: $\forall a \in X\ \mathrm{post}^G(a) \neq \varnothing \wedge \mathrm{post}^G(a) \subseteq R$. This means that $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) = \mathrm{post}^G(X)$ (since $\mathrm{post}^\sharp$ is pointwise) and $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) \subseteq R$. Note that abstract order in $\mathbb{D}_\sharp$ is Hoare order. Since every containment path that stays within R is finite, every element in R can reach an element in X by crossing 0 or more containment-edges, hence $R \sqsubseteq X$, and $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) \sqsubseteq X$. That is, $\gamma_\sharp(X)$ is a recurrent set via Theorem 4.1. Also, since $X \subseteq R$, then $X \sqsubseteq R$, and hence $\gamma_\sharp(R) = \gamma_\sharp(X)$.

(ii) Let $X \in V$ be such that $\forall a \in X.\ \mathrm{post}^G(a) \neq \varnothing$, $\mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, X) = 1$, and also $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) \sqsubseteq_\sharp X$. By properties of $\mathrm{eval}^\sharp$, $\forall a \in \mathrm{post}^\sharp(C_{\mathrm{body}}, X).\ \mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, \{a\}) = 1$. Let $R = X \cup \mathrm{post}^G(X) \subseteq V$. Note the following: (i) every $a \in X$ has a non-empty set of outgoing post-edges, all leading into R; (ii) since $\mathrm{post}^\sharp(C_{\mathrm{body}}, X) \sqsubseteq_\sharp X$, either $R = X = \{\bot_{\mathtt{m}}\}$ or every $a \in \mathrm{post}^G(X)$ has a containment-edge leading into $R \supseteq X$. Hence, R satisfies the definition of recurrent component. □

Lemma 4.4. May-exiting elements cannot be part of a recurrent component.

Proof. Let $a \in V$ be s.t. $\mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, \{a\}) \neq 1$. Then it cannot be part of a recurrent component by definition.

Let $a \in V$ be s.t. $\mathrm{eval}^\sharp(\psi_{\mathrm{ent}}, \{a\}) = 0$. That is, from the properties of $\mathrm{eval}^\sharp$, for every concrete memory state $m \in \gamma_\sharp(\{a\})$, ψ_{ent} does not hold. Since $\psi_{\mathrm{ent}} \vee \varphi_{\mathrm{exit}} = 1$, for every $m \in \gamma_\sharp(\{a\})$, either φ_{exit} holds or $m = \varepsilon$, and it cannot be that $\mathrm{eval}^\sharp(\neg\varphi_{\mathrm{exit}}, \{a\}) = 1$.

Let $a \in V$ be s.t. it would be marked as must-exiting by recursively applying the rules (ii) and (iii) from the definition of must-exiting element. By induction, we can show that for every concrete memory state $m \in \gamma_\sharp(\{a\})$, every concrete path that originates in it, eventually reaches a memory state where ψ_{ent} does not hold (a memory state that must exit the loop or is erroneous). That is, $\gamma_\sharp(\{a\})$ cannot be contained in a recurrent set. From Theorem 4.2, it follows that a cannot be part of a recurrent component, since otherwise $\gamma_\sharp(\{a\})$ would have to be contained in a recurrent set, and we would have a contradiction.

Let $\text{post}^\sharp$ be monotone and let $a \in V$ be such that (a) it would be marked as may-exiting by recursively applying rules (ii) and (iii) from the definition of may-exiting element; and (b) let a belong to a recurrent component $R \subseteq V$. Let us show that this is impossible. From assumption (a), is follows by induction that by taking one or more post- and *reversed* containment-edges a can reach an element $a' \in V$, s.t. a' is must-exiting or s.t. $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, \{a'\}) \neq 1$. From assumption (b) and Theorem 4.2, there is $X \subseteq R$ s.t. $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, X) = 0$, $\text{post}^\sharp(C_{\text{body}}, X) \sqsubseteq_\sharp X$, and $a \subseteq X$. Consider an arbitrary element $a' \in V$ that is reachable from a by taking one or more post- and *reversed* containment-edges. From monotonicity of $\text{post}^\sharp$, it follows by induction that $\{a'\} \sqsubseteq_\sharp X$ and therefore, from the properties of $\text{eval}^\sharp$, $\text{eval}^\sharp(\neg\varphi_{\text{exit}}, \{a'\}) = 1$. Also, by the above development, a' cannot be must-exiting. Note that the corollaries of assumption (b) contradict the corollaries of assumption (a).

From the above, we conclude that it cannot be that a may-exiting element a is a part of a recurrent component. □

Theorem 4.3. For an abstract state graph $G = \langle V, E_p, E_c \rangle$ and some recurrent component $R \subseteq V$, *FindRecComp* produces $C \subseteq V$ such that $C \cup R$ is the maximal recurrent component of G.

Proof. Let $R_{\max} \subseteq V$ be the maximal recurrent component of G, which does exist due to Lemma 4.3.

First, note that $C \cup R$ is indeed a recurrent component. If it was not, in lines 5–7, the elements that make $C \cup R$ violate the definition of recurrent component would be excluded from C (such elements can only be in C, and not in R). That is, $C \cup R \subseteq R_{\max}$.

Next, let us represent the execution of *FindRecComp* as a chain of n approximations $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_{n-1}$ where C_0 is as in line 2, $C_{n-1} = C$ is the output of the procedure,

for $0 \leq i \leq n-2$, $C_{i+1} = C_i \setminus C_i^-$, where

$$C_i^- = \{a \in C_i \mid (\{(a,a') \mid a' \in C_i \cup R\} \cap E_c = \varnothing) \wedge (\mathrm{post}^G(a) = \varnothing \vee \mathrm{post}^G(a) \nsubseteq C_i \cup R)\}$$

And $C_{n-1}^- = \varnothing$. That is C_i^- is the set of elements that make $C_i \cup R$ violate the definition of recurrent component.

Let us by induction prove that $R_{\max} \subseteq C \cup R$. From Lemma 4.4, it follows that $R_{\max} \subseteq C_0 \cup R$ as no may- or must-exiting element can be in a recurrent component. For $0 \leq i \leq n-2$, let us assume that $R_{\max} \subseteq C_i \cup R$. From the definition of recurrent component, for every $a \in R_{\max}$,

$$(\exists a' \in R_{\max}.\ (a,a') \in E_c) \vee (\mathrm{post}^G(a) \neq \varnothing \wedge \mathrm{post}^G(a) \subseteq R_{\max})$$

From the definition of C_i^- above, it follows that $C_i^- \cap R_{\max} = \varnothing$ (that is, no $a \in R_{\max}$ satisfies the condition to be included in C_i^-. Thus we have that $R_{\max} \subseteq C_i \cup R$, and $C_i^- \cap R_{\max} = \varnothing$, and $C_{i+1} = C_i \setminus C_i^-$, hence $R_{\max} \subseteq C_{i+1} \cup R$.

By induction, $R_{\max} \subseteq C \cup R$ and hence $R_{\max} = C \cup R$. □

Chapter 5

Experiments in Finding Recurrent Sets

We have implemented the algorithms of Chapters 3 and 4 as two separate prototype tools. We evaluated the tools using a number of test programs available in the program analysis community.

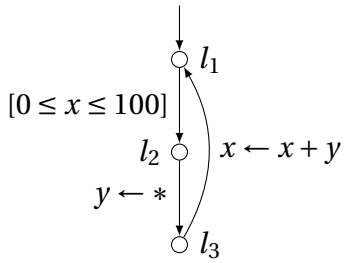
Prototype of Chapter 3

The prototype of Chapter 3 is implemented in Scala. The implementation consists of two main components. The first one performs generic fixed point approximation using chaotic iteration, as discussed in Section 2.3, introduction to Chapter 3, and Appendix 3.B. The computation is parameterized with the abstract domain and uses interation order that prioritizes inner loops (it uses a prioritized worklist and assigns priorities to locations based on their position in hierarchical SCC decomposition [Bou93b]). The second component is an implementation of the trace partitioning domain, as described in Chapter 3. The underlying memory abstract domain is a product of polyhedra (linear inequalities) and linear congruences (constraints asserting divisibility by constants). The implementations of the polyhedral domain and the domain of linear congruences are provided by Parma Polyhedra Library [BHZ08].

As input, the prototype accepts unstructured programs – i.e., graphs with edges labelled by numeric statements – in its own format. The format is similar to the input format of some other tools that work with transition systems, e.g. T2 [Bro+16]. This could allow in future to re-use some of the components of T2 that perform pre-processing of the input, e.g., convert C source code to a transition system. An example of an input program is shown in Fig. 5.1.

```
1 | x, y: Int
2 | l1: [x >= 0, x <= 100] :l2
3 | l2: y = * :l3
4 | l3: x = x + y :l1
```

(a) Input text.



(b) Corresponding program graph.

Figure 5.1: Example of an input program for the implementation of the algorithm of Chapter 3.

Prototype of Chapter 4

The prototype of Chapter 4 actually pre-dates the one of Chapter 3. It is implemented in Java as a library and needs a driver Java program. As input, it accepts individual structured loops that must be constructed using the provided API. An example of a driver program is shown in Fig. 5.2. Lines 1–7 initialize the abstract domain and abstract transformers. Lines 8–14 construct the loop condition and body. Lines 15–17 initialize and run the analysis.

The implementation again consists of two main components. The first one builds and analyses an abstract reachability graph of a loop, as described in Chapter 4, and is parameterized with an abstract domain. The second component implements interval domain which is used to analyse numeric programs, and the domain of 3-valued structures [SRW02] that is used to analyse heap-manipulating programs. The implementation of the domain of 3-valued structures is based on making calls to a modified[1] version of the tool TVLA[2] [LMS04].

Benchmarks

To evaluate the implementations, we used several numeric test programs available in the program analysis community. The prototype of Chapter 4 also supports heap-manipulating programs, but we only evaluated this support using a small number of examples that we produced ourselves; most of them are presented in Section 4.3.

An early benchmark for non-termination analyses of programs was produced by the authors of the tool called Invel[3] [VR08]. The benchmark consists of 55 numeric

[1] The only purpose of the modifications was to allow using TVLA as a library in a Java application.
[2] `http://www.cs.tau.ac.il/~tvla/`, last accessed in May 2016.
[3] `http://www.key-project.org/nonTermination/`, last accessed in May 2016.

```
1 while (x ≥ 1) {
2   if (x = 60) x ← 50;
3   x ← x + 1;
4   if (x = 100) x ← 0;
5 }
```

(a) Informal program text.

```
1  IntISet intSet = new IntISet();
2  IntervalID intervalD = new IntervalID(intSet);
3  BoxID boxD = new BoxID(intervalD);
4  Var x = boxD.makeVar("x");
5  boxD.freeze();
6  IntervalPost intervalPost = new IntervalPost(boxD);
7  StmtPostI stmtPost = new StmtPostI(intervalPost);
8  StmtFacI f = new StmtFacI(intervalD);
9  Formula<IntProp> cond = f.cmp(Cmp.GT, x, 0);
10 Stmt<IntStmt, IntProp>   body = f.seq(
11     f.ifThen(f.cmp(Cmp.EQ, x, 60), f.set(x, 50)),
12     f.set(x, f.op(Op.PLUS, x, 1)),
13     f.ifThen(f.cmp(Cmp.EQ, x, 100), f.set(x, 0))
14 );
15 RecSetAlgI recSetAlg = new RecSetAlgI(boxD, stmtPost);
16 RecSetAlgI.Run run = recSetAlg.makeRun(body, cond, cond.negate()
       , stmtPost.model(cond));
17 run.findAll();
```

(b) Driver program to run the analysis.

Figure 5.2: Example of a driver program for the implementation of the algorithm of Chapter 4.

Java programs and 4 heap-manipulating programs. Of the numeric programs, 53 deterministically fail to terminate for some inputs, one program always terminates and one encodes the Collatz conjecture[4] (thus we do not know whether or not it terminates for all inputs).

Another well-known benchmark is the Competition on Software Verification, also known as SV-COMP[5]. One of the categories in the competition is "termination", which is maintained by the teams of the tools AProVE [Gie+14] and Ultimate Büchi Automizer [Hei+16], and consists of more than 300 C programs. Most of the offered programs terminate for all inputs (which the tools entering the competition need to prove), but some have non-terminating behaviours.

To evaluate our implementations, we used 53 numeric programs from Invel benchmark and selected 44 non-terminating[6] numeric programs from the 2015 edition of SV-COMP[7]. We manually converted those programs to the input formats of out prototype implementations (for the prototype of Chapter 4 we converted only 18 programs out of 44).

It is worth noting one of the first benchmarks for termination and non-termination provers – the Termination Competition[8]. Historically, it focuses on termination of rewriting systems, but since 2009 includes a category for Java bytecode programs, and since 2014 – for C programs and integer transition systems. We did not use this benchmark in our evaluation, but to our knowledge the category of C programs considerably overlaps with SV-COMP.

Summary of the Results

In Table 5.1, we summarize the experimental results[9] and compare our prototype implementations to 3 existing tools that can prove non-termination of programs: AProVE [Gie+14], Ultimate Büchi Automizer [Hei+16], and HipTNT+ [LQC15]. We did not evaluate these tools ourselves, but just repeat the results that we found elsewhere. In par-

[4] The conjecture can be summarized as follows. Take a positive integer n. If n is even, divide it by 2. If n is odd, multiply it by 3 and add 1 (i.e., obtain $3n+1$). Repeat this process indefinitely. The conjecture is that this process always reaches 1. It has been tested for starting values of n up to 2^{60}, but no formal proof is known.

[5] `http://sv-comp.sosy-lab.org/`, last accessed in May 2016.

[6] In the sense that they fail to terminate for some inputs.

[7] `http://sv-comp.sosy-lab.org/2015/`, last accessed in May 2016.

[8] `http://termination-portal.org/wiki/Termination_Competition`, last accessed in May 2016.

[9] We give a detailed table in Appendix 5.A.

ticular, for Ultimate Büchi Automizer and HipTNT+, we do not have the results for Invel programs. For AProVE, we give results for Invel programs as reported by [Bro+11], and for SV-COMP programs – as reported by the 2015 edition of Termination Competition[10] (the version of AProVE that participated in 2015 edition of SV-COMP did not include a non-termination prover for C programs).

The table should be read as follows. For our prototypes, column "OK" is the number of programs for which a prototype could find a recurrent set. The sets were later checked *manually* for reachability. Most test programs consist of a single loop and a stem that gives initial values to program variables; and to check reachability, we only needed to intersect the inferred recurrent set with the produced set of initial states. Column"M" is the number of programs that originally fall outside of the class that a prototype can handle, but after we introduced small modifications (e.g., replaced a non-linear condition with an equivalent linear one), a prototype would find a recurrent set. Column "U" is the number of programs for which no recurrent set could be found due to technical limitations of a prototype. In particular, our implementations do not support arrays, pointers, recursion, some instances of division and modular arithmetic, etc. We note that the prototype of Chapter 4 was tested only on 18 of 44 SV-COMP non-terminating programs. The remaining 26 programs contribute to the "U" column. Column "X" is the number of programs that are formally not affected by the limitations of the prototypes, but for which no recurrent set could be found.

For the other tools, the columns "OK" and "X" give the number of programs for which the tools were able, and respectively failed to prove non-termination. Column "U" gives the number of programs for which we did not find reported results.

Numbers in brackets compare the tools against the prototype of Chapter 3. These are the numbers of test programs, for which the prototype of Chapter 3 produces the opposite outcome (fails to find a recurrent set in a program that was marked as non-terminating and vice versa). For example, for the SV-COMP benchmark, AProVE successfully proved non-termination for 30 programs. For 6 of those, the prototype of Chapter 3 could not find recurrent set. On the other hand, AProVE could not prove non-termination of 10 programs, and for 6 of those, the prototype of Chapter 3 found a recurrent set

[10] `http://www.termination-portal.org/wiki/Termination_Competition_2015`, last accessed in May 2016.

Table 5.1: Summary of the experimental results.

		Chapter 3				Chapter 4			AProVE			Ultimate		HipTNT+	
	Tot.	OK	M	U	X	OK	U	X	OK	U	X	OK	X	OK	X
Invel	53	46	5	2	-	39(+1)	3(+3)	11(+11)	51(+2)	-	2(+2)	-	-	-	-
SV-C	44	32	-	9	3	10(+1)	3(+3)	31(+20)	30(+6)	4	10(+6)	37(+11)	7(+6)	35(+7)	9(+4)

Note that Table 5.1 should not be interpreted as a *direct* comparison of our prototypes and 3 listed tools, as they prove different things about the programs. The prototype of Chapter 3 finds existential recurrent sets; the prototype of Chapter 4 finds universal recurrent sets; AProVE and Ultimate Büchi Automizer prove the existence of at least one non-terminating execution; and HipTNT+ to our knowledge proves that from some initial states, all executions are non-terminating.

Discussion

In numeric benchmarks, the prototype of Chapter 4 is not a good match for the specialized numeric tools. A possible explanation is that the prototype lacks two mechanisms that are important for the analysis of numeric programs: relational reasoning and extrapolation.

The prototype of Chapter 4 uses interval domain that tracks possible ranges of variable values, but not the relations between them, and as a result cannot find recurrent sets that take the form as in 'If x is greater than y, then the loop will not terminate'. We found that out of 14 Invel benchmarks, where the prototype could not find a recurrent set, at least 7 require a relational domain. Also, most numeric analysis based on abstract interpretation use widening to make informed guesses about the values of fixed points. At the same time, the prototype of Chapter 4 lacks any similar extrapolation technique and can only produce a recurrent set from abstract states that appear directly as a result of post-condition operation. Because of that, the analysis cannot handle programs like the one in Example 3.3, where it needs to guess a non-deterministic choice (of a variable's value) that causes a program to not terminate.

The prototype of Chapter 3 was implemented after the prototype of Chapter 4 and addresses both shortcomings. It uses polyhedral domain and employs lower widening to guess the limits of descending chains. As a result, it performs on par with other numeric tools, and there is no *single* dominant cause of failure of the analysis (i.e., failure to find a recurrent set in a program that has non-terminating behaviours). Most fail-

```
1 | while (x > 0)
2 |   x ← −2x + 9
```

Figure 5.3: A program where a recurrent set corresponds to a fixed point of some mathematical function.

ures are due to the lack of support of certain language features in the analysis: some forms of division, arrays, pointers, dynamic memory, and recursion. The former four are a matter of implementation effort, and they see rather limited use in the benchmarks. Handling recursion may be much trickier, as we have not yet looked into formulating our analyses for the inter-procedural setting.

Actually, there is a class of non-terminating behaviours that both prototypes do not handle well. An example of a problematic program is shown in Fig. 5.3. The recurrent set in this example is $x = 3$, which corresponds to a fixed point of the function $\lambda x. -2x + 9$. If we allow x to be rational and use backward analysis to built for this program a chain of approximations of a recurrent set, standard widening techniques fail to find its stable limit (which should be $x = 3$). Some other tools have dedicated techniques to handle such cases (e.g., AProVE can detect behaviours where important variables do not change in a loop iteration). There is hope that in the framework of abstract interpretation such cases can be handled with specialized fixed point computation techniques (e.g., with policy iteration [Cos+05]).

5.A Detailed Experimental Results

Table 5.2: Detailed experimental results.

Test name	Chapter 3	Chapter 4
Invel		
alternatingIncr	OK	OK
alternDiv	OK	OK
alternDivWide	OK	OK
alternDivWidening	OK	X
alternKonv	OK	OK
complInterv	M	OK
complInterv2	OK	OK
complInterv3	OK	OK
complxStruc	OK	X
convLower	OK	OK
cousot	OK	OK
doubleNeg	M	X
even	OK	OK
ex01	OK	OK
ex02	OK	OK
ex03	OK	OK
ex04	OK	OK
ex05	OK	OK
ex06	OK	OK
ex07	OK	OK
ex08	OK	OK
ex09half	U	X
factorial	M	X
fib	OK	X
flip	OK	OK

flip2	OK	X
gauss	OK	OK
gcd	OK	OK
lcm	OK	X
marbie1	OK	OK
marbie2	OK	OK
middle	OK	X
mirrorInterv	OK	X
mirrorIntervSim	OK	OK
moduloLower	M	OK
moduloUp	M	OK
narrowing	OK	OK
narrowKonv	OK	OK
plait	U	OK
sunset	OK	OK
trueDiv	OK	OK
twoFloatInterv	OK	OK
upAndDown	OK	OK
upAndDownIneq	OK	OK
whileBreak	OK	U
whileIncr	OK	OK
whileIncrPart	OK	OK
whileNested	OK	U
whileNestedOffset	OK	U
whilePart	OK	OK
whileSingle	OK	OK
whileSum	OK	X
whileTrue	OK	OK

SV-COMP

Arrays02-EquivalentConstantIndices	U	U
Division	U	OK
Madrid	OK	OK
NonTermination1	OK	OK
NonTermination2	OK	U
NonTermination3	U	U
NonTermination4	OK	OK
NonTerminationSimple2	OK	OK
NonTerminationSimple3	OK	X
NonTerminationSimple4	OK	OK
NonTerminationSimple5	OK	X
NonTerminationSimple6	OK	OK
NonTerminationSimple7	OK	OK
NonTerminationSimple8	OK	U
NonTerminationSimple9	OK	X
RecursiveNonterminating	U	U
Rotation180	OK	OK
WhileTrue	OK	OK
BradleyMannaSipma-CAV2005-Fig1-modified	OK	U
ChenCookFuhsNimkarOHearn-TACAS2014-Introduction	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex2.02	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex2.03	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex2.04	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex2.05	U	U
ChenFlurMukhopadhyay-SAS2012-Ex2.06	X	U
ChenFlurMukhopadhyay-SAS2012-Ex2.11	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex2.12	X	U
ChenFlurMukhopadhyay-SAS2012-Ex2.14	X	U
ChenFlurMukhopadhyay-SAS2012-Ex2.15	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex2.17	OK	U

ChenFlurMukhopadhyay-SAS2012-Ex3.02	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex3.06	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex3.08	OK	U
ChenFlurMukhopadhyay-SAS2012-Ex4.01	OK	U
HarrisLalNoriRajamani-SAS2010-Fig2	OK	U
HenzingerJhalaMajumdarSutre-POPL2002-LockingExample	OK	U
LeikeHeizmann-WST2014-Ex5	OK	U
LeikeHeizmann-WST2014-Ex6	OK	U
Urban-WST2013-Fig1	OK	U
Velroyen	OK	U
joey	U	U
PodelskiRybalchenko-2004VMCAI-Ex2-alloca	U	U
Urban-2013WST-Fig1-alloca	U	U
Velroyen-alloca	U	U

Chapter 6

Recurrent Sets in Analysis for Sufficient Pre-Conditions

In this chapter, we demonstrate how the notion of recurrent set can be used in an analysis for safety and propose a novel approach for computing weakest liberal safe preconditions of programs. Our goal will be to compute (an under-approximation of) the set of safe states of a program, from which *no* execution can lead to a failure (such as violating an assertion, dividing by zero, or dereferencing a dangling-pointer – i.e., an event that causes program execution to abort and signal an error).

First, let us reiterate and make explicit some observations that we could make over the course of the previous chapters. Forward static analyses (based on post-condition computation) usually compute program invariants that hold of executions starting from given initial conditions, e.g., over-approximations of reachable states. And conversely, backward static analyses (based on the computation of pre-conditions or predecessors) for universal properties compute program invariants that ensure given assertions hold of all executions, e.g., under-approximations of safe states or recurrent sets. Forward analysis of programs has been a notable success, while backward analysis has seen much less research and is done less frequently[1]. The standard formulation of forward analyses is based on over-approximating a least fixed point of a transfer function (or over-approximating a solution to a recursive system of equations) that represents the forward semantics of a program. Conversely, backward analyses for uni-

[1]There are good examples of numeric backward analyses though. Apart from the analysis of Chapter 3, one notable example of an under-approximate backward analysis is [Min13]. There has also been some success in inferring piecewise-linear ranking functions with backward analysis [UM15].

versal properties usually involve under-approximating a greatest fixed point.

Over-approximating abstractions used by forward analyses are more common and well-developed than the under-approximations used by backward analyses, and this was one of the motivations behind the research that we described in Chapters 3 and 4. Another indirect approach to under-approximation is via over-approximate abstraction and under-approximate complementation which we will denote with an overbar: $\overline{(\cdot)}$. In this setting, if one wants to compute an under-approximation of safe program states, they instead compute an over-approximation of unsafe states and then take its complement. However, computing a complement is, in many cases, infeasible or impractical (including, for 3-valued structures, separation logic, or polyhedra). In this chapter, we suggest an approach to backward analysis that replaces complementation with *under-approximate logical subtraction* operation (it can also be understood as *and with complement* or *not implies*). In a nutshell, our approach characterizes the set of safe program states as a least fixed-point above a recurrent set. Soundness of the abstract computation is then ensured by subtracting an over-approximation of the unsafe states.

Using subtraction instead of complementation has several advantages. First, it is easier to define in powerset domains, for which complementation can be hard or impractical. Second, as the approximations of safe and unsafe states are the results of analyzing the same code, they are strongly related and so subtraction may be more precise than a general under-approximate complementation.

Our approach is not restricted to a specific abstract domain and we use it to analyze numeric examples (using the domain of intervals) and examples coming from shape analysis (using the domain of 3-valued structures).

6.1 Background

Let us be given a concrete domain $\mathscr{L}_\flat$ and an abstract domain $\mathbb{D}_\sharp$. In particular, in this chapter, we analyze structured programs, and the concrete domain is the domain of memory states (since we are analysing structured programs, and the locations of a corresponding unstructured program are implicit in the analysis). The analysis works in a memory abstract domain, where elements represent sets of memory states. Since

the memory abstract domain is the main domain of the analysis in this chapter, we again denote it by $\mathbb{D}_\sharp$ instead of $\mathbb{D}_\mathtt{m}$.

Complementation

For the abstract domain $\mathbb{D}_\sharp$, we define *complementation* to be a function $\overline{(\cdot)} : \mathbb{D}_\sharp \to \mathbb{D}_\sharp$ that for every $d \in \mathbb{D}_\sharp$, produces another element with a disjoint concretization:

$$\gamma_\sharp(\overline{d}) \sqcap_\flat \gamma_\sharp(d) = \bot_\flat$$

That is, an abstract element $d \in \mathbb{D}_\sharp$ and its complement $\overline{d}$ represent disjoint sets of (program or memory) states; but we do not require that $\gamma_\sharp(\overline{d}) \sqcup_\flat \gamma_\sharp(l) = \top_\flat$. For example, if $d \in \mathbb{D}_\sharp$ over-approximates the unsafe states, then $\gamma(d)$ under-approximates the safe states. In a domain that is a power set of a set of atomic elements, we can use standard set-theoretic complement.

Subtraction

We define *subtraction* as a function $(\cdot - \cdot) : \mathbb{D}_\sharp \times \mathbb{D}_\sharp \to \mathbb{D}_\sharp$, such that for $d_1, d_2 \in \mathbb{D}_\sharp$,

$$\gamma_\sharp(d_1 - d_2) \sqsubseteq_\flat \gamma_\sharp(d_1)$$
$$\gamma_\sharp(d_1 - d_2) \sqcap_\flat \gamma_\sharp(d_2) = \bot_\flat$$

We claim that a useful subtraction is often easier to define than a useful complementation. For example, given some underlying domain $\mathbb{D}$, we can define a coarse but still useful subtraction for the *power set domain* $\mathbb{D}_\sharp = \mathscr{P}(\mathbb{D})$ in the following way: for $D_1, D_2 \in \mathbb{D}_\sharp$,

$$D_1 - D_2 = \{d_1 \in D_1 \mid \forall\, d_2 \in D_2.\ \gamma(d_1) \sqcap_\flat \gamma(d_2) = \bot\} \tag{6.1}$$

In particular, when elements of $\mathbb{D}$ represent sets of memory states,

$$D_1 - D_2 = \{d_1 \in D_1 \mid \forall\, d_2 \in D_2.\ \gamma(d_1) \cap \gamma(d_2) = \varnothing\}$$

This way, subtraction can be defined, e.g., in the domain of 3-valued structures that does not readily support complementation. This definition is suitable for numeric domains as well. For example, in the interval domain, with this definition $\{[1;3],[5;7]\} -$

$\{[6;8]\} = \{[1;3]\}$.

We also note that for every $d_0 \in \mathbb{D}_\sharp$, the function $\lambda d.(d_0 - d)$ is a complementation. However, for a given d, the accuracy of this complement depends on the actual choice of d_0. One can think that later in this chapter, an over-approximation of the set of unsafe memory states will be d, and some approximation of the set of safe memory states will be d_0.

Input Language

We define the analysis for a subset of the language of structured programs [2], similar to the one we used in Chapter 4. We require that in our programs, assumption statements appear only at the start of a branch or at the entry or exit of a loop (they cannot be used as normal atomic statements):

$$C ::= a \mid C_1 \,;\, C_2 \mid ([\varphi] \,;\, C_1) + ([\psi] \,;\, C_2) \mid ([\psi] \,;\, C)^* \,;\, [\varphi]$$

and branch and loop guard assumptions are exhaustive: $\varphi \vee \psi = 1$. Additionally (we did not have this constraint before) we require that there are no nested loops.

Non-Error Memory States

Let us denote the set of non-error memory states by

$$\mathbb{M}_{\setminus\varepsilon} = \mathbb{M} \setminus \{\varepsilon\}$$

Largely, this chapter focuses specifically on non-error memory states. In particular, when talking about the set of unsafe-memory states, we will be interested in the set of *non-error* unsafe memory states. The error memory state itself is trivially unsafe excluding the error memory state from the computation saves us from needing to represent it in the abstract domain. Thus, we can think of the concrete domain of our analysis as $\mathscr{L}_\flat = \mathscr{P}(\mathbb{M}_{\setminus\varepsilon})$.

Example 6.1. The program in Fig. 6.1 will be our running example for this chapter. Fig. 6.1a shows program text in pseudocode, and Fig. 6.1b shows the corresponding formal structured program (recall that we considered a similar program in Example 4.1).

[2] The reader is again invited to re-visit Section 2.5 before proceeding.

```
1  while (x ≥ 1) {
2    if (x = 60) {
3      x ← 50;
4    }
5    x ← x + 1;
6    if (x = 100) {
7      x ← 0;
8    }
9  }
10 assert(0)
```

(a) Informal text in pseudocode.

$$
\begin{aligned}
&{}_1\big([x \geq 1];\\
&\quad {}_2(([x = 60];{}_3 x \leftarrow 50) + ([x \neq 60]; \mathtt{skip}));\\
&\quad {}_5 x \leftarrow x + 1;\\
&\quad {}_6(([x = 100];{}_7 x \leftarrow 0) + ([x \neq 100]; \mathtt{skip}));\\
&\big)^*; [x \leq 0];\\
&{}_{10}\mathtt{assert}(0)
\end{aligned}
$$

(b) Formal structured program.

Figure 6.1: Example program 6.1.

In the formal program, we label the statements that are important for the analysis with the corresponding line numbers from Fig. 6.1a (like in ${}_3 x \leftarrow 50$).

6.2 Fixpoint Characterizations of Safe and Unsafe States

Let us define two auxiliary transformers, in addition to those defined in (2.2) of Section 2.3.1. For a statement C and a set of non-error memory states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$, let

$$\mathrm{fail}(T_{\mathbb{M}}(C)) = \{m \in \mathbb{M}_{\setminus\varepsilon} \mid (m, \varepsilon) \in T_{\mathbb{M}}(C)\}$$

$$\mathrm{pre{+}fail}(T_{\mathbb{M}}(C), M) = \mathrm{pre}(T_{\mathbb{M}}(C), M) \cup \mathrm{fail}(T_{\mathbb{M}}(C))$$

That is, $\mathrm{fail}(T_{\mathbb{M}}(C))$ is the set of non-error memory states from which C fails.

Lemma 6.1. For a statement C and a set of non-error memory states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$

$$\mathrm{wp}(T_{\mathbb{M}}(C), M) = \mathbb{M}_{\setminus\varepsilon} \setminus \mathrm{pre{+}fail}(T_{\mathbb{M}}(C), \mathbb{M}_{\setminus\varepsilon} \setminus M)$$

Proof. The proof is a direct calculation based on the definitions. We present it in full in Appendix 6.A. □

For a structured program C, our goal is to compute (an under-approximation of) $\mathrm{wp}(T_{\mathbb{M}}(C), \mathbb{M}_{\setminus\varepsilon})$ and (an over-approximation of) its complement – $\mathrm{fail}(T_{\mathbb{M}}(C))$. If we

are interested in termination with specific postcondition θ, we can add an $\texttt{assert}(\theta)$ statement to the end of the program. We characterize these sets (as is standard [Cla77; Cou81]) as solutions to two functionals P and N that associate a statement C and a set of states M (resp., V) $\subseteq \mathbb{M}_{\setminus\varepsilon}$ with a predicate $P(C,M)$, resp., $N(C,V)$. $P(C,M)$, which we call the *positive side*, denotes the states that must either lead to successful termination in M or cause non-termination. Conversely, $N(C,V)$, which we call the *negative side*, denotes the states that may lead to failure or termination in V. Given a program statement (where a denotes an atomic statement) and $M, V \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$\begin{aligned}
&P(a,M) = \mathrm{wp}(T_{\mathbb{M}}(a), M) && N(a,V) = \text{pre+fail}(T_{\mathbb{M}}(a), V)\\
&P([\theta],M) = [\![\neg\theta]\!] \cup M && N([\theta],V) = [\![\theta]\!] \cap V\\
&P(\texttt{assert}(\theta),M) = [\![\theta]\!] \cap M && N(\texttt{assert}(\theta),V) = [\![\neg\theta]\!] \cup V\\
&P(C_1\,;C_2,M) = P(C_1,P(C_2,M)) && N(C_1\,;C_2,V) = N(C_1,N(C_2,V))\\
&P(C_1+C_2,M) = P(C_1,M) \cap P(C_2,M) && N(C_1+C_2,V) = N(C_1,V) \cup N(C_2,V)\\
&P(C^*,M) = \mathrm{gfp}_{\subseteq}\,\lambda X.M \cap P(C,X) && N(C^*,V) = \mathrm{lfp}_{\subseteq}\lambda Y.V \cup N(C,Y)
\end{aligned} \tag{6.2}$$

Lemma 6.2. For a statement C and set of states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$

$$P(C,M) = \mathbb{M}_{\setminus\varepsilon} \setminus N(C, \mathbb{M}_{\setminus\varepsilon} \setminus M)$$

Proof. The proof is by structural induction. We present it in full in Appendix 6.A. □

Lemma 6.3. For a statement C and sets of states $M, V \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$P(C,M) = \mathrm{wp}(T_{\mathbb{M}}(C), M)$$
$$N(C,V) = \text{pre+fail}(T_{\mathbb{M}}(C), V)$$

Proof. The proof is by structural induction. We present it in full in Appendix 6.A. □

P and N may seem redundant as at this point they are essentially shortcuts for wp and pre+fail. The difference is that P and N are defined by induction over the statement structure (and for loops – via fixed points), and for the standard transformers, induction and fixed points are hidden in the definition of the transition relation of a compound statement. Still, we found that P and N benefit the exposition. Later, we

will introduce the *abstract* positive and negative sides $P^\flat$ and $N^\sharp$, which will have a different structure from the standard abstract transformers, but similarly to P and N, they will be defined by induction over the statement structure, using limits of ascending chains for loops. Thus, the reader will be able to draw parallels between the abstract and concrete positive and negative sides (while the association to the standard transformers becomes less clear in the abstract case).

6.3 Least Fixed-Point Characterization of Safe States

To compute an abstract positive side directly, we would have to under-approximate a greatest fixed point. As discussed before, this can be problematic since most domains are geared towards over-approximating least fixed points. Hence, instead, we restate the problem for loops such that the resulting characterization is based on a least fixed point. In this section, we focus on the looping statement:

$$C_{\text{loop}} = ([\psi_{\text{ent}}]\,;\, C_{\text{body}})^*\,;\, [\varphi_{\text{exit}}] \tag{6.3}$$

where C_{body} is the *loop body*; if ψ_{ent} holds, the execution may enter the loop body; and if φ_{exit} holds the execution may exit the loop. To simplify the presentation, in what follows, we assume that the input-output relation of C_{body} is directly known. Since (by our assumption) C_{body} is itself loop-free, $T_{\mathbb{M}}(C_{\text{body}})$ does not induce fixed points, and the transformers for the loop body can be obtained by combining the transformers for its sub-statements.

Recurrent Sets

In what follows, will reformulate the characterizations of safe states in terms of least fixed points above recurrent sets.

For the loop in (6.3), similarly to Chapter 4 (see Section 4.1), let us define a *projection of universal recurrent set on the loop entry*. This is a set $R_\forall$, s.t.

$$\begin{aligned} &R_\forall \subseteq [\![\neg\varphi_{\text{exit}}]\!] \\ &\forall m \in R_\forall.\, \big(\forall m' \in \mathbb{M}.\, (m, m') \in T_{\mathbb{M}}(C_{\text{body}}) \Rightarrow m' \in R_\forall\big) \end{aligned} \tag{6.4}$$

Intuitively, if an execution (of the corresponding unstructured program) reaches the loop entry in one of these memory states, it will stay in the loop forever. In the rest of the chapter, we will call this set a *universal recurrent set of a loop*.

Similarly, an *existential recurrent set of the loop* is a set $R_\exists$, s.t.

$$R_\exists \subseteq [\![\psi_{\text{ent}}]\!]$$
$$\forall m \in R_\exists.\ \exists m' \in R_\exists.\ (m, m') \in T_{\mathbb{M}}(C_{\text{body}})$$

Intuitively, if an execution reaches the loop entry in a memory states $m \in R_\exists$, it may stay in the loop forever. Indeed. First, the execution the loop from that state. Second, from Lemma 2.8, there exists a terminating execution of the loop body that leads to a memory state m' that also belongs to $R_\exists$. This way, we can construct from m a non-terminating execution postfix.

Example 6.1 (continued)**.** Recall, that Example 4.1 considers a program that contains the same loop, as in our running example. For that loop, the analysis of Chapter 4 finds a universal recurrent set $R_\forall = (1 \le x \le 60) \vee (x \ge 101)$. In our experiments, we also used the tool E-HSF to find recurrent sets of numeric programs. With three calls to E-HSF using different recurrent set templates[3], we could infer the recurrent set $R_\forall = (4 \le x \le 60) \vee (x \ge 100)$. In the rest of the running example, we will use this recurrent set.

Lemma 6.4. For the loop C_{loop} in (6.3), s.t.

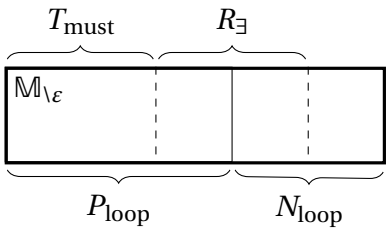
$$C_{\text{loop}} = ([\psi_{\text{ent}}]\,;\, C_{\text{body}})^*\,;\, [\varphi_{\text{exit}}]$$

and a set of states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$,

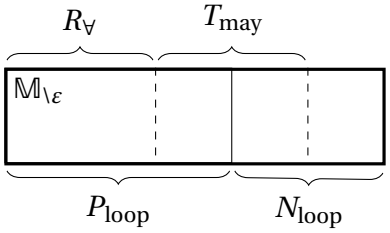
$$R_\forall \subseteq P(C_{\text{loop}}, M)$$
$$R_\exists \setminus N(C_{\text{loop}}, \mathbb{M}_{\setminus\varepsilon} \setminus M) \subseteq P(C_{\text{loop}}, M)$$

Proof idea. For $R_\exists$, the result follows from Lemma 6.2. For $R_\forall$, one can show that a universal recurrent set is always below $P(C_{\text{loop}}, \varnothing)$ which is below $P(C_{\text{loop}}, M)$ for every $M \subseteq \mathbb{M}_{\setminus\varepsilon}$. We present the full proof in Appendix 6.A. □

[3] To our knowledge, E-HSF internally uses Farkas lemma, and it needs the user to provide a template for the result – a parameterized formula, for which E-HSF will infer the values of the parameters.



(a) Partitioning with existential recurrence.



(b) Partitioning with universal recurrence.

Figure 6.2: Partitioning of the states at the loop entry.

Positive Least Fixed Point via Recurrent Sets

We begin with an informal explanation of how we move from a greatest fixed point formulation to a least fixed point one. Observe that for the loop in (6.3), the positive and negative sides are characterized as follows. For $M, V \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$\begin{aligned} P(C_{\text{loop}}, M) &= \text{gfp}_{\subseteq}\, \lambda X.([\![\neg\varphi_{\text{exit}}]\!] \cup M) \cap \big([\![\neg\psi_{\text{ent}}]\!] \cup P(C_{\text{body}}, X)\big) \\ N(C_{\text{loop}}, V) &= \text{lfp}_{\subseteq} \lambda Y.[\![\varphi_{\text{exit}}]\!] \cap V) \cup \big([\![\psi_{\text{ent}}]\!] \cap N(C_{\text{body}}, Y)\big) \end{aligned} \tag{6.5}$$

Then, since loops only occur at the top level, a program C_{prg} that contains the loop C_{loop} can be expressed as C_{init} ; C_{loop} ; C_{rest}, where C_{init} or C_{rest} may be `skip`. Let:

(i) $P_{\text{rest}} = P(C_{\text{rest}}, \mathbb{M}_{\setminus\varepsilon})$ be the safe states of the loop's continuation.

(ii) $N_{\text{rest}} = N(C_{\text{rest}}, \varnothing)$ be states that may cause failure of the loop's continuation. Note that $N_{\text{rest}} = \mathbb{M}_{\setminus\varepsilon} \setminus P_{\text{rest}}$.

(iii) $P_{\text{loop}} = P(C_{\text{loop}}, P_{\text{rest}})$ be the safe states of the loop and its continuation.

(iv) $N_{\text{loop}} = N(C_{\text{loop}}, N_{\text{rest}})$ be states that may cause failure of the loop or its continuation. Note that $N_{\text{loop}} = \mathbb{M}_{\setminus\varepsilon} \setminus P_{\text{loop}}$.

For the loop in (6.3), Fig. 6.2 shows how the states entering the loop can be partitioned. In the figure, by T_{must}, we denote the states that must cause successful termination *of the loop* (in a state belonging to P_{rest}) and by T_{may}, we denote states that may cause successful termination.

Fig. 6.2a shows that the positive side for the loop in (6.3) can be partitioned into the following two parts:

(i) $R_\exists \setminus N_{\text{loop}}$ – states that may cause non-termination but may not fail;

(ii) T_{must} – states that must cause successful termination of the loop.

T_{must} can be characterized as the least fixed point:

$$T_{\text{must}} = \text{lfp}_{\subseteq} \lambda X.([\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}}) \cup \Big(([\![\psi_{\text{ent}}]\!] \cap P_{\text{rest}}) \cup [\![\neg\varphi_{\text{exit}}]\!]\Big) \cap \text{wp}(C_{\text{body}}, X)\Big)$$

Intuitively, the states in $[\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}}$ cause the loop to immediately terminate (such that the rest of the program does not fail), those in

$$(([\![\psi_{\text{ent}}]\!] \cap P_{\text{rest}}) \cup [\![\neg\varphi_{\text{exit}}]\!]) \cap \text{wp}(C_{\text{loop}}, [\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}})$$

can make one iteration through the loop, and so on.

Fig. 6.2b shows that the positive side can also be partitioned in another way:

(i) $R_{\forall}$ – states that must cause non-termination of the loop;

(ii) $T_{\text{may}} \setminus N_{\text{loop}}$ – states that may cause successful termination but may not fail.

In a way similar to [Cou81], T_{may} can be characterized as the least fixed point:

$$T_{\text{may}} = \text{lfp}_{\subseteq} \lambda X.([\![\varphi_{\text{exit}}]\!] \cap P_{\text{rest}}) \cup \big([\![\psi_{\text{ent}}]\!] \cap \text{pre}(C_{\text{body}}, X)\big)$$

Intuitively, from states $[\![\varphi_{\text{exit}}]\!] \cap P_{\text{rest}}$, the loop *may* immediately terminate in a state safe for C_{rest}; from states $[\![\psi_{\text{ent}}]\!] \cap \text{pre}(C_{\text{body}}, [\![\varphi_{\text{exit}}]\!] \cap P_{\text{rest}})$ the loop may make one iteration and terminate, and so on. From this, it can be shown that

$$T_{\text{may}} \setminus N_{\text{loop}} = \text{lfp}_{\subseteq} \lambda X.\big(([\![\varphi_{\text{exit}}]\!] \cap P_{\text{rest}}) \setminus N_{\text{loop}}\big) \cup \big(([\![\neg\varphi_{\text{exit}}]\!] \cap \text{pre}(C_{\text{body}}, X)) \setminus N_{\text{loop}}\big)$$

We replace ψ_{ent} with $\neg\varphi_{\text{exit}}$, since the states in $[\![\psi_{\text{ent}}]\!] \cap [\![\varphi_{\text{exit}}]\!] \cap \text{pre}(C_{\text{body}}, X)$ are either already included in the first disjunct (if belonging to P_{rest}), or are unsafe and removed by subtraction.

Following these least fixed point characterizations, we can re-express the equation for the positive side of the loop (6.5) using the existential recurrent set $R_{\exists}$ as follows, where $N = N(C_{\text{loop}}, \mathbb{M}_{\setminus\varepsilon} \setminus M)$. For $M \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$P^{\exists}(C_{\text{loop}}, M) = \text{lfp}_{\subseteq} \lambda X.\,(R_{\exists} \setminus N) \cup ([\![\neg\psi_{\text{ent}}]\!] \cap M) \cup \Big(([\![\psi_{\text{ent}}]\!] \cap M) \cup [\![\neg\varphi_{\text{exit}}]\!]\Big) \cap \text{wp}(C_{\text{body}}, X)\Big) \tag{6.6}$$

or using the universal recurrent set $R_\forall$ as follows:

$$P^\forall(C_{\text{loop}}, M) = \text{lfp}_\subseteq \lambda X.\, R_\forall \cup \left((\llbracket \varphi_{\text{exit}} \rrbracket \cap M) \setminus N\right) \cup \left(\left(\llbracket \neg \varphi_{\text{exit}} \rrbracket \cap \text{pre}(C_{\text{body}}, X)\right) \setminus N\right) \tag{6.7}$$

Theorem 6.1. The alternative characterizations of the positive side of the loop: (6.6) and (6.7) – under-approximate the original characterization (6.5). That is, for $M \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$P^\exists(C_{\text{loop}}, M) \subseteq P(C_{\text{loop}}, M)$$
$$P^\forall(C_{\text{loop}}, M) \subseteq P(C_{\text{loop}}, M)$$

Proof. We present the proof in Appendix 6.A. □

6.4 Approximate Characterizations

In Sections 6.2 and 6.3, we characterized both the negative and the positive sides as least fixed points. For the negative side, our goal is to over-approximate the least fixed point, and we can do that using standard tools. That is, we move to the abstract memory domain $\mathbb{D}_\sharp$ ordered by $\sqsubseteq_\sharp$, with the least element $\bot_\sharp$, greatest element $\top_\sharp$, join $\sqcup_\sharp$, widening $\triangledown_\sharp$ and concretization $\gamma_\sharp : \mathbb{D}_\sharp \to \mathscr{P}(\mathbb{M}_{\setminus\varepsilon})$. In the previous sections we made sure that the error memory state appears neither on the positive nor on the negative side. Thus, we do not need to represent error in the abstract memory domain.

As before, we assume that abstract transformers for loop bodies are given. First, we assume that we are given over-approximate versions of pre, fail and of an assumption operation (an operation that approximates intersection with the set of memory states that satisfy a formula) For a loop-free statement C, $d \in \mathbb{D}_\sharp$, and a memory state formula θ,

$$\gamma_\sharp(\text{pre}^\sharp(C, d)) \supseteq \text{pre}(T_\mathbb{M}(C), \gamma_\sharp(d))$$
$$\gamma_\sharp(\text{fail}^\sharp(C)) \supseteq \text{fail}(T_\mathbb{M}(C))$$
$$\text{pre+fail}^\sharp(C, d) \supseteq \text{pre+fail}(T_\mathbb{M}(C), d)$$
$$\gamma_\sharp([\theta, d]^\sharp) \supseteq \llbracket \theta \rrbracket \cap \gamma_\sharp(d)$$

We abbreviate $[\theta, \top_\sharp]^\sharp$ as $[\theta]^\sharp$.

Additionally, we assume that we are also given under-approximate versions of meet $\sqcap_\flat$, wp and of an assumption operation, s.t. for a loop-free statement C, $d, d_1, d_2 \in \mathbb{D}_\sharp$, and a memory state formula θ,

$$\gamma_\sharp(d_1 \sqcap_\flat d_2) \subseteq \gamma_\sharp(d_1) \cap \gamma_\sharp(d_2)$$
$$\gamma_\sharp(\mathrm{wp}^\flat(C, d)) \subseteq \mathrm{wp}(T_{\mathbb{M}}(C), d)$$
$$\gamma_\sharp([\theta, d]^\flat) \subseteq [\![\theta]\!] \cap \gamma_\sharp(d)$$

We abbreviate $[\theta, \top_\sharp]^\flat$ as $[\theta]^\flat$.

Later in this section, we will reduce the number of under-approximate operations that we need and obtain an analysis where subtraction is the only under-approximate operation that we require.

Approximate Positive and Negative Sides

Negative Side Given a statement C and $n \in \mathbb{D}_\sharp$, the approximate negative side $N^\sharp(C, n)$, which over-approximates $N(C, \gamma_\sharp(n))$, is non-recursively defined by induction over the statement structure:

$$\begin{aligned}
N^\sharp(a, n) &= \text{pre+fail}^\sharp(a, n), \text{ for } a \in \mathbb{A} \\
N^\sharp(C_1\,;\, C_2, n) &= N^\sharp(C_1, N^\sharp(C_2, n)) \\
N^\sharp(([\varphi]; C_1) + ([\psi]; C_2), n) &= [\varphi, N^\sharp(C_1, n)]^\sharp \sqcup_\sharp [\psi, N^\sharp(C_2, n)]^\sharp \\
N^\sharp((([\psi_{\text{ent}}]; C_{\text{body}})^*; [\varphi_{\text{exit}}], n) &= \text{the first } n_j \in \{n_i\}_{i \geq 0}, \text{ such that } n_{j+1} \sqsubseteq_\sharp n_j \text{ where} \\
&\qquad n_0 = [\varphi, n]^\sharp \text{ and } n_{i+1} = n_i \triangledown_\sharp (n_i \sqcup_\sharp [\psi, N^\sharp(C_{\text{body}}, n_i)]^\sharp)
\end{aligned}$$

Example 6.1 (continued)**.** Let us continue with our running example (the program in Fig. 6.1). First, let us assume that program variables (just x in this case) take *integer* values. For the abstract domain, we use disjunctive refinement over intervals allowing a bounded number of disjuncts[4] (e.g., via [BHZ07]). In the examples, by $\langle x : [a; b], y :$

[4]This analysis pre-dates the one of Chapter 3 and does not use trace partitioning, although it may benefit from it. Our prototype implementation for this chapter chooses which disjuncts to merge (to meet the bound) based on some notion of a distance between the hypercubes (we prefer to not go into the details here). With trace partitioning, we will merge the disjuncts based on the future paths through the program which may be more desirable.

$[c;d]\rangle$ we will denote a singleton abstract state of a program with two variables x and y, representing the set of concrete states, satisfying $(a \le x \le b) \wedge (c \le y \le d)$.

For this abstract domain and the formulas, appearing in the program, $[\cdot,\cdot]^\sharp$ and $[\cdot,\cdot]^\flat$ coincide, and we write $[\cdot,\cdot]$ to denote either. To emphasize that the analysis can produce useful results even when using a coarse subtraction function, we use subtraction as defined in (6.1). That is, we just drop from the positive side those disjuncts that have a non-empty intersection with the negative side. For example, $\{\langle x : [1;3]\rangle, \langle x : [5;7]\rangle\} - \langle x : [6;8]\rangle = \langle x : [1;3]\rangle$. The analysis is performed mechanically by a prototype tool that we have implemented.

To simplify the presentation, in this example, we bound the number of disjuncts in a domain element by 2. Also to simplify the presentation, we omit the $\sharp$- and $\flat$-superscripts, and write, e.g., pre+fail for pre+fail$^\sharp$. For a statement labeled with i, we write N_i^j to denote the result of the j-th step of the computation of its negative side, and N_i to denote the computed value (similarly, for P).

Let us start with the analysis of the negative side, going backwards from the end of the program. For the final statement,

$$N_{10}^1 = \text{pre+fail}(\texttt{assert}(0), \bot) = \top$$

then, we proceed to the first approximation for the loop (for clarity, we compute pre of the body in steps),

$$N_1^1 = [x \le 0, N_{10}^1] = \langle x : (-\infty; 0]\rangle$$
$$N_7^1 = \text{pre+fail}(x \leftarrow 0, N_1^1) = \top$$
$$N_6^1 = [x = 100, N_7^1] \sqcup [x \ne 100, N_1^1] = \{\langle x : (-\infty; 0]\rangle, \langle x : [100]\rangle\}$$
$$N_5^1 = \text{pre+fail}(x \leftarrow x + 1, N_6^1) = \{\langle x : (-\infty; -1]\rangle, \langle x : [99]\rangle\}$$
$$N_3^1 = \text{pre+fail}(x \leftarrow 50, N_5^1) = \bot$$
$$N_2^1 = [x = 60, N_3^1] \sqcup [x \ne 60, N_5^1] = \{\langle x : (-\infty; -1]\rangle, \langle x : [99]\rangle\}$$
$$N_1^2 = N_1^1 \sqcup [x \ge 1, N_2^1] = \{\langle x : (-\infty; 0]\rangle, \langle x : [99]\rangle\}$$

then, repeating the same similar sequence of steps for the second time gives

$$N_1^2 = N_1^2 \sqcup [x \ge 1, N_2^2] = \{\langle x : (-\infty; 0]\rangle, \langle x : [98, 99]\rangle\}$$

at which point we detect an unstable bound. The choice of widening strategy is not our focus here, and for demonstration purposes, we proceed without widening, which allows to discover the stable bound of 61. In a real-world tool, to retain precision, some form of widening *up to* [HPR97] or landmarks [SK06] could be used. Thus, we take

$$N_1 = \{\langle x : (-\infty; 0]\rangle, \langle x : [61; 99]\rangle\}$$
$$N_2 = \{\langle x : (-\infty; -1]\rangle, \langle x : [61; 99]\rangle\}$$
$$N_3 = \bot$$
$$N_5 = \{\langle x : (-\infty; -1]\rangle, \langle x : [61; 99]\rangle\}$$
$$N_6 = \{\langle x : (-\infty; 0]\rangle, \langle x : [61; 100]\rangle\}$$
$$N_7 = \top$$
$$N_{10} = \top$$

Positive Side For a statement C and a pair of disjoint elements $p, n \in \mathbb{D}_\sharp$ (i.e., $\gamma_\sharp(p) \cap \gamma_\sharp(n) = \varnothing$), we define the approximate positive side $P^\flat(C, p, n)$, which under-approximates $P(C, \mathbb{M}_{\setminus\varepsilon} \setminus \gamma_\sharp(n))$. We define $P^\flat(C, p, n)$ mutually with an auxiliary $Q^\natural(C, p, n)$ by induction on the structure of C. Intuitively, $Q^\natural(C, p, n)$ is an abstraction of $P(C, \gamma_\sharp(p))$, which may not be under-approximate due to the use of over-approximate operations (e.g., join and widening). Also, note how n is used to represent the complement of the set of interest.

For non-looping statements, $P^\flat$ and $Q^\natural$ are non-recursively defined as follows:

$$P^\flat(C, p, n) = Q^\natural(C, p, n) - N^\sharp(C, n)$$
$$Q^\natural(a, p, n) = \mathrm{wp}^\flat(a, p), \text{ for } a \in \mathbb{A}$$
$$Q^\natural(C_1 \,;\, C_2, p, n) = P^\flat(C_1, P^\flat(C_2, p, n), N^\sharp(C_2, n))$$
$$Q^\natural(([\varphi]; C_1) + ([\psi]; C_2), p, n) = \left(P^\flat(C_1, p, n) \sqcap_\flat P^\flat(C_2, p, n)\right) \sqcup_\sharp [\neg\psi, P^\flat(C_1, p, n)]^\flat \sqcup_\sharp [\neg\varphi, P^\flat(C_2, p, n)]^\flat$$

For a loop $C_{\text{loop}} = ([\psi_{\text{ent}}]; C_{\text{body}})^*; [\varphi_{\text{exit}}]$, let us define a sequence $\{q_i\}_{i \geq 0}$ of approximants to $Q^\natural(C_{\text{loop}}, p, n)$, where $q_{i+1} = q_i \triangledown_\sharp (q_i \sqcup_\sharp \tau(q_i))$, and the initial point q_0 and the transformer τ are defined following either the characterization (6.6) using an approxi-

mation $R^\flat_\exists \in \mathbb{D}_\sharp$ of an existential recurrent set of the loop:

$$q_0 = \left(R^\flat_\exists - N^\sharp(C_{\text{loop}}, n)\right) \sqcup_\sharp [\neg\psi, p]^\flat$$
$$\tau(q_i) = \left([\psi_{\text{ent}}, p]^\flat \sqcap_\sharp \text{wp}^\flat(C_{\text{body}}, q_i)\right) \sqcup_\sharp [\neg\varphi_{\text{exit}}, \text{wp}^\flat(C_{\text{body}}, q_i)]^\flat$$

or following (6.7) using an approximation $R^\flat_\forall \in \mathbb{D}_\sharp$ of a universal recurrent set:

$$q_0 = R^\flat_\forall \sqcup_\sharp \left([\varphi_{\text{exit}}, p]^\flat - N^\sharp(C_{\text{loop}}, n)\right)$$
$$\tau(q_i) = \left([\neg\varphi_{\text{exit}}, \text{pre}^\sharp(C_{\text{body}}, q_i)]^\flat - N^\sharp(C_{\text{loop}}, n)\right)$$

As for loop-free commands, $Q^\natural$ can be computed first, and $P^\flat$ defined using the result. That is, we can define $Q^\natural(C_{\text{loop}}, p, n) = q_j$, where q_j is the first element such that $q_{j+1} \sqsubseteq_\sharp q_j$, and then define $P^\flat(C_{\text{loop}}, p, n) = Q^\natural(C_{\text{loop}}, p, n) - N^\sharp(C_{\text{loop}}, n)$.

Alternatively, $P^\flat$ and $Q^\natural$ can be computed simultaneously by defining a sequence $\{p_i\}_{i\geq 0}$ of safe under-approximants of $P^\flat(C_{\text{loop}}, p, n)$, where $p_0 = q_0 - N^\sharp(C_{\text{loop}}, n)$ and $p_{i+1} = (p_i \sqcup_\sharp (p_i \triangledown_\sharp \tau(q_i))) - N^\sharp(C_{\text{loop}}, n)$. Then $P^\flat(C_{\text{loop}}, p, n) = p_j$, where p_j is the first element such that $q_{j+1} \sqsubseteq_\sharp q_j$ or $p_{j+1} \not\sqsubseteq_\sharp p_j$. In this case, we may obtain a sound $P^\flat$ before the auxiliary $Q^\natural$ has stabilized. While we have not yet done rigorous experimental validation, we prefer this approach when dealing with coarse subtraction.

When analyzing a top-level program C_{prg}, the analysis starts with $N^\sharp(C_{\text{prg}}, \bot_\sharp)$ and precomputes $N^\sharp$ (an over-approximation of unsafe states) for all statements of the program. Then it proceeds to compute $P^\flat(C_{\text{prg}}, \top_\sharp, \bot_\sharp)$ (an under-approximation of safe input states) reusing the precomputed results for $N^\sharp$.

We are using over-approximate join and widening on the positive side, and $Q^\natural$ may not under-approximate the positive side of the concrete characterization. Widening allows the ascending chain to converge, and subtraction of the negative side ensures soundness of $P^\flat$. In other words, the concrete characterizations (6.6) and (6.7) are used to *guide* the definition of the approximate characterizations, but soundness is argued directly rather than by using (6.6) and (6.7) as an intermediate step.

Theorem 6.2. For a statement C and $p, n \in \mathbb{D}_\sharp$ s.t. $\gamma_\sharp(p) \cap \gamma_\sharp(n) = \varnothing$

$$N^\sharp(C, n) \supseteq N(C, \gamma_\sharp(n))$$
$$P^\flat(C, p, n) \subseteq P(C, \mathbb{M}_{\setminus\varepsilon} \setminus \gamma_\sharp(n))$$

As a result, for a top-level program C_{prg},

$$\gamma_\sharp(N^\sharp(C_{\text{prg}}, \bot_\sharp)) \supseteq N(C_{\text{prg}}, \varnothing)$$

(i.e., it over-approximates input states that may lead to failure), and

$$\gamma_\sharp(P^\flat(C_{\text{prg}}, \top_\sharp, \bot_\sharp)) \subseteq P(C_{\text{prg}}, \mathbb{M}_{\setminus\varepsilon})$$

(i.e., it under-approximates safe input states)

Proof idea. The argument for $N^\sharp$ proceeds in a standard way for over-approximate computations. Soundness for $P^\flat$ then follows due to the use of subtraction. We give the full proof in Appendix 6.A. □

Optimizations of Constraints

Use of over-approximate operations Since we are subtracting $N^\sharp(C, n)$ anyway, we can relax the right-hand side of the definition of $Q^\natural(C, p, n)$ without losing soundness. Specifically, we can replace under-approximating and must- operations by their over-approximating and may- counterparts. This way, we obtain an analysis where subtraction is the only under-approximating operation.

(i) For a loop-free statement C, use $\text{pre}^\sharp(C, p)$ in place of $\text{wp}^\flat(C, p)$ (note that we *already* use $\text{pre}^\sharp$ on the positive side for loop bodies when starting from a universal recurrent set). This can be handy, e.g., for power set domains where $\text{pre}^\sharp$ (unlike $\text{wp}^\flat$) can be applied element-wise. Also, these transformers may coincide for deterministic loop-free statements (if the abstraction is precise enough). Later, when discussing Example 6.2, we note some implications of this substitution.

(ii) For a memory state formula θ, use $[\theta, \cdot]^\sharp$ in place of $[\theta, \cdot]^\flat$. Actually, for some combinations of an abstract domain and a language of formulas, these transformers coincide. For example, in a polyhedral domain, conjunctions of linear constraints have precise representations as domain elements.

(iii) For branching statements, use $[\varphi, P^\flat(C_1, p, n)]^\sharp \sqcup_\sharp [\psi, P^\flat(C_2, p, n)]^\sharp$ in place of the original expression.

(iv) In the definition of $Q^\natural$, an over-approximate meet operation $\sqcap_\sharp$ suffices.

The result of these relaxations is:

$$Q^\sharp(a,p,n) = \mathrm{pre}^\sharp(a,p), \text{ for } a \in \mathbb{A}$$
$$Q^\sharp(C_1\,;C_2,p,n) = P^\flat(C_1,P^\flat(C_2,p,n),N^\sharp(C_2,n))$$
$$Q^\sharp(([\varphi];C_1)+([\psi];C_2),p,n) = [\varphi,P^\flat(C_1,p,n)]^\sharp \sqcup_\sharp [\psi,P^\flat(C_2,p,n)]^\sharp$$

$$q_0 = \left(R^\sharp_\exists - N^\sharp(C_{\text{loop}},n)\right) \sqcup_\sharp [\neg\psi_{\text{ent}},p]^\sharp$$
$$\tau(q_i) = \left([\psi_{\text{ent}},p]^\sharp \sqcap_\sharp \mathrm{pre}^\sharp(C_{\text{body}},q_i)\right) \sqcup_\sharp [\neg\varphi_{\text{exit}},\mathrm{pre}^\sharp(C_{\text{body}},q_i)]^\sharp$$

or

$$q_0 = R^\sharp_\forall \sqcup_\sharp \left([\varphi_{\text{exit}},p]^\sharp - N^\sharp(C_{\text{loop}},n)\right)$$
$$\tau(q_i) = \left([\neg\varphi_{\text{exit}},\mathrm{pre}^\sharp(C_{\text{body}},q_i)]^\sharp - N^\sharp(C_{\text{loop}},n)\right)$$

No Subtraction for $Q^\sharp$ For a similar reason, subtraction can be removed from the characterization of $Q^\sharp$ without affecting soundness of $P^\flat$.

Bound on the Positive Side Another observation is that for a loop C_{loop} as in (6.3), the positive side $P(C_{\text{loop}},M)$ is bounded by $[\![\neg\varphi_{\text{exit}}]\!] \sqcup M$, as can be seen from the characterization (6.5). This can be incorporated into a specialized definition for loops, defining $P^\flat(C_{\text{loop}},p,n) = \left(Q^\sharp(C_{\text{loop}},p,n) \sqcup_\sharp ([\neg\varphi_{\text{exit}}]^\sharp \sqcup_\sharp p)\right) - N^\sharp(C_{\text{loop}},n)$ or by performing the meet during computation of $Q^\sharp$ by defining $q_{i+1} = \left(q_i \triangledown_\sharp (q_i \sqcup_\sharp \tau(q_i))\right) \sqcap_\sharp ([\neg\varphi_{\text{exit}}]^\sharp \sqcup_\sharp p)$.

Example 6.1 (continued)**.** Let us now show how the computation of the positive side works for our running example. Recall that earlier we decided to initialize the computation of the positive side with a universal recurrent set $R_\forall = \{\langle x : [4;60]\rangle, \langle x : [100;+\infty)\rangle\}$. In this example, universal recurrence and safety coincide, and our analysis will be able to improve the result by showing that the states in $\langle x : [1;3]\rangle$ are also safe. Since we are using a power set domain, we choose to use pre instead of wp for the all statements, not just for the loop (where we need to use it due to using a universal recurrent set). We start with

$$P^1_{10} = \mathrm{pre}(\texttt{assert}(0),\top) - N_{10} = \bot - N_{10} = \bot$$

then proceed to the loop (again, computing pre of its body in steps),

$$
\begin{aligned}
P_1^1 &= R_\forall \sqcup [x \le 0, P_{10}^1] - N_1 = \{\langle x : [4;60]\rangle, \langle x : [100;+\infty)\rangle\} \\
P_7^1 &= \mathrm{pre}(x \leftarrow 0, P_1^1) - N_7 = \bot \\
P_6^1 &= [x = 100, P_7^1] \sqcup [x \neq 100, P_1^1] - N_6 \\
&= \{\langle x : [4:60]\rangle, \langle x : [101;+\infty)\rangle\} - N_6 \\
&= \{\langle x : [4:60]\rangle, \langle x : [101;+\infty)\rangle\} \\
P_5^1 &= \mathrm{pre}(x \leftarrow x+1, P_6^1) - N_5 = \{\langle x : [3:59]\rangle, \langle x : [100;+\infty)\rangle\} \\
P_3^1 &= \mathrm{pre}(x \leftarrow 50, P_5^1) - N_3 = \top \\
P_2^1 &= [x = 60, P_3^1] \sqcup [x \neq 60, P_5^1] - N_2 \\
&= \{\langle x : [3;59]\rangle, \langle x : [60]\rangle, \langle x : [100;+\infty)\rangle\} - N_2 \\
&= \{\langle x : [3;60]\rangle, \langle x : [100;+\infty)\rangle\} \\
P_1^2 &= (P_1^1 \sqcup ([x \ge 1, P_2^1] - N_2)) - N_2 = \{\langle x : [3;60]\rangle, \langle x : [100;+\infty)\rangle\}
\end{aligned}
$$

at which point we detect an unstable bound, but we again proceed without widening and are able to discover the stable bound of 1. Also note that, P_1 is bounded by $P_{10} \sqcup [\neg(x \le 0)] = \langle x : [1;+\infty)\rangle$. This bound could be used to improve the result of widening. Thus, we take

$$
\begin{aligned}
P_1 &= \{\langle x : [1;60]\rangle, \langle x : [100;+\infty)]\rangle\} \\
P_2 &= \{\langle x : [0;60]\rangle, \langle x : [100;+\infty)]\rangle\} \\
P_3 &= \top \\
P_5 &= \{\langle x : [0;59]\rangle, \langle x : [100;+\infty)]\rangle\} \\
P_6 &= \{\langle x : [1;60]\rangle, \langle x : [101;+\infty)]\rangle\} \\
P_7 &= \bot \\
P_{10} &= \bot
\end{aligned}
$$

Thus, in this example, our analysis was able to prove that initial states $\{\langle x : [1;60]\rangle, \langle x : [100;+\infty)]\rangle\}$ are safe, which is a slight improvement over the output of E-HSF.

6.4.1 Approximating a Recurrent Set

When approximating the positive side for a loop, the computation is initialized with an approximation of a recurrent set. Our analysis is able to start with either an existential or a universal recurrent set depending on what search procedure is available for the domain. The instantiation of our approach for numerical domains may use the algorithms of Chapters 3 and 4[5]. The instantiation for shape analysis with 3-valued logic may use the algorithm of Chapter 4. Normally, the search procedures are incomplete: the returned sets are under-approximate, and the search itself might not terminate (we assume the use of timeouts in this case). This incompleteness leaves room for our analysis to improve the approximation. For example, sometimes a solver produces a universal recurrent set that is closed under forward transformers, but is not closed under backward ones. In such cases, our analysis can produce a larger recurrent set.

6.5 Examples

In this section, we demonstrate our approach on several additional examples: first for a numeric domain, and then for the shape analysis domain of 3-valued structures. We note that numeric programs are considered here solely for the purpose of clarity of explanation, since the domain is likely to be familiar to most readers. We do not claim novel results specifically for the analysis of numeric programs, although we note that our approach may be able to complement existing tools.

Example 6.1 aims at describing steps of the analysis in detail. Example 6.2 includes a pragmatic discussion on using $\text{pre}^\sharp$ on the positive side. Examples 6.3 and 6.4 consider programs from a shape analysis domain.

Example 6.2. In this example, we consider the program in Fig. 6.3. In the program, $*$ stands for a value non-deterministically chosen at runtime. All the assumptions made for Example 6.1 are in effect for this one as well, except that we increase the bound on the size of the domain element to 4.

[5]For numeric programs, the original implementation of the analysis of this chapter was using the tool E-HSF [BPR13] that is capable of approximating both existential and universal recurrence. For heap-manipulating programs, we used a prototype procedure that was, in a way, a predecessor of the analysis of Chapter 4.

```
1  while (x ≥ 1) {
2    if (x ≤ 99) {
3      if (y ≤ 0 ∧ *) {
4
5        assert(0)
6
7      }
8      if (*) {
9        x ← −1
10     }
11   }
12   x ← x + 1
13 }
14 assert(y ≠ 0)
```

(a) Informal program in pseudocode.

$$
\begin{aligned}
&{}_1([x \geq 1];\\
&\quad {}_2(([x \leq 99];\\
&\qquad {}_3(([y \leq 0];{}_4({}_5\mathtt{assert}(0) + \mathtt{skip}))\\
&\qquad\quad + ([y \geq 1];\mathtt{skip}));\\
&\qquad {}_8({}_9x \leftarrow -1 + \mathtt{skip})\\
&\quad) + ([x \geq 100];\mathtt{skip}));\\
&\quad {}_{12}x \leftarrow x + 1\\
&)^*;[x \leq 0];\\
&{}_{14}\mathtt{assert}(y \neq 0)
\end{aligned}
$$

(b) Formal structured program.

Figure 6.3: Example program 6.2.

Again, we start with the negative side. For the final location,

$$N^1_{14} = \text{pre+fail}(\mathtt{assert}(y \neq 0), \bot) = \langle x : \top, y : [0] \rangle$$

then, proceed to the loop

$$N^1_1 = [x \leq 0, N^1_{14}] = \langle x : (-\infty, 0], y : [0] \rangle$$

$$N^1_{12} = \text{pre+fail}(x \leftarrow x + 1, N^1_1) = \langle x : (-\infty; -1], y : [0] \rangle$$

$$N^1_9 = \text{pre+fail}(x \leftarrow -1, N^1_{12}) = \langle x : \top, y : [0] \rangle$$

$$N^1_8 = N^1_9 \sqcup N^1_{12} = \langle x : \top, y : [0] \rangle$$

$$N^1_5 = \top$$

$$N^1_3 = [y \leq 0, N^1_5] \sqcup N^1_8 = \langle x : \top, y : (-\infty; 0] \rangle$$

$$N^1_2 = [x \leq 99, N^1_3] \sqcup [x \geq 100, N^1_{12}] = \langle x : (-\infty; 99], y : (-\infty; 0] \rangle$$

$$N^2_1 = N^1_1 \sqcup [x \geq 1, N^1_2] = \{\langle x : (-\infty, 0], y : [0] \rangle, \langle x : [1; 99], y : (-\infty; 0] \rangle\}$$

repeating the steps gives

$$N^3_1 = \{\langle x : (-\infty, 0], y : [0] \rangle, \langle x : [1; 99], y : (-\infty; 0] \rangle\} = N^2_1$$

Thus, we take

$$N_1 = \{\langle x : (-\infty, 0], y : [0]\rangle, \langle x : [1; 99], y : (-\infty; 0]\rangle\}$$

$$N_2 = \langle x : (-\infty; 99], y : (-\infty; 0]\rangle$$

$$N_3 = \langle x : \top, y : (-\infty; 0]\rangle$$

$$N_5 = \top$$

$$N_8 = \{\langle x : \top, y : [0]\rangle, \langle x : [0; 98], y : (-\infty; 0]\rangle\}$$

$$N_9 = \langle x : \top, y : [0]\rangle$$

$$N_{12} = \{\langle x : (-\infty; 0], y : [0]\rangle, \langle x : [0; 98], y : (-\infty; 0]\rangle\}$$

$$N_{14} = \langle x : \top; y : [0]\rangle$$

To initialize the positive side, we can use a universal recurrent set produced by the procedure of Chapter 4. The result is $R_\forall = \langle x : [100; +\infty), y : \top\rangle$.

Again, we choose to use pre for all the computation steps on the positive side.

$$P^1_{14} = \text{pre}(\texttt{assert}(y \neq 0), \top) - N_{14} = \{\langle x : \top, y : (-\infty; -1]\rangle, \langle x : \top, y : [1; +\infty)\rangle\}$$

$$P^1_1 = R_\forall \sqcap (D_{x \leq 0} \sqcap P^1_{15}) - N^1_1 =$$

$$\{\langle x : [100; +\infty), y : \top\rangle, \langle x : (-\infty; 0], y : (-\infty; -1]\rangle, \langle x : (-\infty; 0], y : [1; +\infty)\rangle\}$$

Before we proceed to the loop body, we need to make a remark on using the combination of pre and subtraction on the positive side. When an abstract program is non-deterministic (because of non-determinism of a concrete program or coarseness of abstraction), pre is often larger than wp, and coarse subtraction can turn it into $\bot$. Consider a fragment of the current example in Fig. 6.4a. The trivial translation to our input language is shown in Fig. 6.4b: having $y \leq 0$ allows to execute the assertion (and fail), but it is always possible to skip the assertion. If we try to analyze the fragment of Fig. 6.4b in isolation, using pre for the positive side, we get the following

$$N_5 = \top$$

$$P_5 = \bot$$

$$N_3 = [y \leq 0, N_5] \sqcup (\top \sqcap \bot) = \langle y : (-\infty; 0]\rangle$$

$$P_3 = [y \leq 0, P_5] \sqcup (\top \sqcap P_\omega) - N_3 = \top - N_3$$

If we use a course subtraction of (6.1), we get

$$P_3 = \bot$$

That is, in this case, we lose all of the positive side, even though there are input states for which the program fragment is safe. The problem here is that the precondition for safety ($y \geq 1$) never had a chance to materialize, and (because of the use of pre) $\top$ easily got into P_3. To work around this in our simple example, we use the following tricks[6]. First, we translate conditions with $*$ into nested conditions, as in Fig. 6.4c or 6.4d. This allows for $[y \geq 1]$ to appear in the equations. Second, for *some* steps of the computation, we allow for a domain element D to be redundant, i.e., to contain such disjuncts $d_1, d_2 \in D$ that $d_1 \sqsubseteq d_2$. Note that widening operators for power sets may require that the domain elements are not redundant [BHZ07], and we would have to remove redundancy before applying widening. Then, from the fragment in Fig. 6.4d, we get the following (for Fig. 6.4c, the steps are almost the same):

$$N_5 = \top$$

$$P_5 = \bot$$

$$N_4 = ([y \leq 0, N_5] \sqcup [y \geq 1, \bot]) = \langle y : (-\infty; 0] \rangle$$

$$P_4 = ([y \leq 0, P_5] \sqcup [y \geq 1, \top]) - N_4 = \langle y : [1; +\infty) \rangle$$

$$N_3 = \bot \sqcup N_4 = \langle y : (-\infty; 0] \rangle$$

$$P_3 = (\top \sqcup P_4) - N_3 = (\top \sqcup \langle y : [1; +\infty) \rangle) - N_3 = \langle y : [1; +\infty) \rangle$$

This way, we were able to show that the fragment is safe for the states in $\langle y : [1; +\infty) \rangle$.

Equipped with these tricks, we return to the example (actually, here we prefer the scheme in Fig. 6.4c to handle the condition in line 3). Recall that

$$P_1^1 = \{\langle x : [100; +\infty), y : \top \rangle, \langle x : (-\infty; 0], y : (-\infty; -1] \rangle,$$
$$\langle x : (-\infty; 0], y : [1; +\infty) \rangle\}$$

[6] If we extend the prototype with trace partitioning, we will still benefit from spliting the conditions, but keeping reduntant elements is useful exactly because we lack trace partitioning here.

```
3 | if (y ≤ 0 ∧ *) {
4 |
5 |   assert(0)
6 |
7 | }
```

(a) Informal program.

$_3(([y \leq 0];_5 \texttt{assert}(0)) + \texttt{skip})$

(b) Single branching.

$_3(([y \leq 0];_4(_5\texttt{assert}(0) + \texttt{skip}))$
$+([y \geq 1];\texttt{skip}))$

(c) Nested branching (1).

$_3(_4(([y \leq 0];_5\texttt{assert}(0)) + ([y \geq 1];\texttt{skip}))$
$+\texttt{skip})$

(d) Nested branching (2).

Figure 6.4: Representations of non-deterministic branching.

then

$$P^1_{12} = \{\langle x : [99;+\infty), y : \top\rangle, \langle x : (-\infty;-1], y : (-\infty;-1]\rangle,$$
$$\langle x : (-\infty;-1], y : [1;+\infty)\rangle\}$$
$$P^1_9 = \{\langle x : \top, y : (-\infty;-1]\rangle, \langle x : \top, y : [1;+\infty)\rangle\}$$
$$P^1_8 = \langle x : \top, y : [1;+\infty)\rangle$$
$$P^1_5 = \bot$$
$$P^1_3 = \langle x : \top, y : [1;+\infty)\rangle$$
$$P^1_2 = \{\langle x : (-\infty;99], y : [1;+\infty)\rangle, \langle x : [100;+\infty), y : \top\rangle\}$$
$$P^2_1 = \{\langle x : [100;+\infty), y : \top\rangle, \langle x : (-\infty;0], y : (-\infty;-1]\rangle,$$
$$\langle x : (-\infty;0], y : [1;+\infty)\rangle, \langle x : [1;99], y : [1;+\infty)\rangle\}$$

then, if we continue in the same way, we get

$$P^3_1 = P^2_1$$

thus, we take

$$P_1 = \{\langle x : [100;+\infty), y : \top\rangle, \langle x : (-\infty;0], y : (-\infty;-1]\rangle,$$
$$\langle x : (-\infty;0], y : [1;+\infty)\rangle, \langle x : [1;99], y : [1;+\infty)\rangle\}$$

Which is the final result.

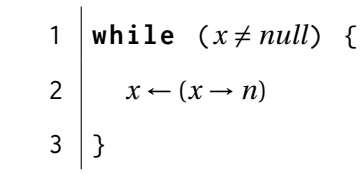


Figure 6.5: Example program 6.3.

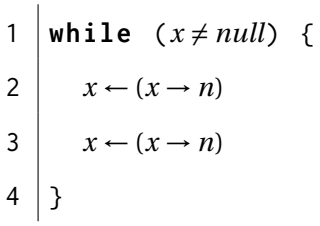


Figure 6.6: Example program 6.4.

Shape Analysis Examples

In what follows, we demonstrate our approach for a shape analysis domain. We treat two simple examples using the domain of 3-valued structures, and we claim that our approach provides a viable decomposition of backward analysis for this domain and probably for some other shape analysis domains.

In Appendix 2.B, we gave a brief description of shape analysis with 3-valued logic that is sufficient to understand the examples. For more information on shape analysis with 3-valued logic, please refer to Sagiv et al. [SRW02] and related papers [RSL10; Arn+06; LMS04].

Example 6.3. In this example, we consider the program in Fig. 6.5 that traverses its input structure in a loop.

For this program, we make the analysis track an additional predicate h_n that states whether the cell has a successor via n-edge. To find a recurrent set, we use the procedure of Chapter 4 that reports that the loop does not terminate when given a cyclic list as input. The output of the procedure consists of various cyclic lists shapes, and one of them is shown in Fig. 6.7[7].

The computation of the positive side will be initialized with this recurrent set, plus the set of structures that immediately exit the loop (without visiting the body), i.e., structures where x does not point to a node. Then, the analysis summarizes all the predecessors of such structures. This results in a number of additional shapes, one of which is shown in Fig. 6.8 and represents an acyclic list of the length 3 or more.

Eventually, the analysis identifies that both cyclic, lasso-shaped, and acyclic lists are safe inputs for the program.

[7] In the figure, predicate r_x denotes reachability via n-edges from the variable x. Binary reachability is not shown for clarity.

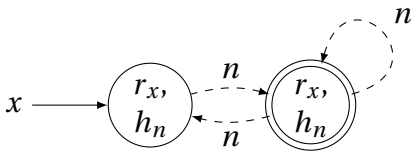


Figure 6.7: A lasso-shaped list. Example of a safe structure causing non-termination of Example 6.3.

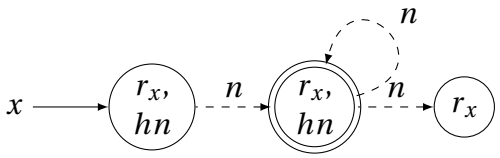


Figure 6.8: A non-cyclic list. Example of a safe structure leading to successful termination of Example 6.3.

Example 6.4. In this example, we consider the program in Fig. 6.6. In this program, the loop body makes two steps through the list instead of just one. While the first step (at line 2) is still guarded by the loop condition, the second step (at line 3) is a source of failure. That is, the program fails when given a list of odd length as an input. The abstraction that we employ is actually not expressive enough to encode such constraints on the length of the list. For example, the produced summary of the negative side (for location 1) contains, e.g., the structure in Fig. 6.8 that represents lists of length just 3 or more, both even and odd As a result, the analysis is able to show that cyclic lists represent safe inputs (as cyclic lists will not appear on the negative side), but the only acyclic list that the analysis identifies as safe is the list of length exactly two.

In this example, we can see precision loss resulting from using too coarse abstraction. In this example, the analysis was not able to summarize the precondition for successful termination (as expected), but still was able to produce the summary of the states that cause non-termination without failure.

6.6 Related Work

In [LA+07], a backward shape analysis with 3-valued logic is presented that relies on the correspondence between 3-valued structures and first-order formulas [Yor+07]. It finds an over-approximation of states that may lead to failure, and then (as 3-valued structures do not readily support complementation) the structures are translated to an equivalent quantified first-order formula, which is then negated. This corresponds to approximating the negative side in our approach and then taking the complement, with the exception that the result is not represented as an element of the abstract domain (though, at least in principle, one could use the symbolic abstraction $\hat{\alpha}$ of [RSY04] to map back to the abstract domain).

For shape analysis with separation logic, preconditions can be inferred using a form of abductive reasoning called bi-abduction [Cal+11]. The analysis uses an over-approximate abstraction, and it includes a filtering step that checks generated preconditions (by computing their respective postconditions) and discards the unsound ones. The purpose of the filtering step – keeping soundness of a precondition produced with over-approximate abstraction – is similar to our use of the negative side.

For numeric programs, the problem of finding preconditions for safety has seen some attention lately.

In [PC13], a numeric program analysis is presented that is based primarily on over-approximation. It simultaneously computes the representations of two sets: of states that may lead to successful termination, and of states that may lead to failure. Then, meet and generic negation are used to produce representations of states that cannot fail, states that must fail, etc.

An under-approximating backward analysis for the polyhedral domain is presented in [Min13]. The analysis defines the appropriate under-approximate abstract transformers and to ensure termination, proposes a lower widening based on the generator representation of polyhedra.

With E-HSF [BPR13], the search for preconditions can be formulated as solving $\forall\exists$ quantified Horn clauses extended with well-foundedness conditions.

6.7 Chapter Conclusion

In this Chapter, we observed how the notion of recurrent set can be applied in an analysis that infers sufficient pre-conditions for safety. More specifically, we decomposed backward analysis into multiple sub-problems: the computations of the positive (potentially safe states) and the negative (definitely unsafe states) sides, and taking difference of the results.

On one hand, this decomposition allowed us to implement backward analysis for the domain of 3-valued structures. On the other hand, we demonstrated how we could start with a problem that has a greatest fixed point formulation and re-state it as a least fixed point above a recurrent set. In a sense, we replaced the approximation of a more general greatest fixed point of backward transformers with the approximation of an ar-

guably less general property – a recurrent set, for which, as we saw in Chapters 3 and 4, we can come up with specialized procedures, not necessary based on backward analysis. In future, this might provide a recipe for using abstract interpretation in verification of more complicated properties (e.g., for temporal model checking of programs).

6.A Omitted Proofs

Lemma 6.1. For a statement C and a set of non-error memory states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$

$$\mathrm{wp}(T_{\mathbb{M}}(C), M) = \mathbb{M}_{\setminus\varepsilon} \setminus \text{pre+fail}(T_{\mathbb{M}}(C), \mathbb{M}_{\setminus\varepsilon} \setminus M)$$

Proof. First, note that pre-condition and predecessors of a set of non-error memory states is also non-error. Then,

$$\begin{aligned}
&\mathbb{M}_{\setminus\varepsilon} \setminus \text{pre+fail}(T_{\mathbb{M}}(C), \mathbb{M}_{\setminus\varepsilon} \setminus M) \\
&= \mathbb{M}_{\setminus\varepsilon} \setminus \big(\mathrm{pre}(T_{\mathbb{M}}(C), \mathbb{M}_{\setminus\varepsilon} \setminus M) \cup \mathrm{fail}(T_{\mathbb{M}}(C))\big) \\
&= \mathbb{M}_{\setminus\varepsilon} \setminus (\{m \in \mathbb{M}_{\setminus\varepsilon} \mid \exists m' \in \mathbb{M}_{\setminus\varepsilon} \setminus M.\ (m, m') \in T_{\mathbb{M}}(C)\} \cup \{m \in \mathbb{M}_{\setminus\varepsilon} \mid (m, \varepsilon) \in T_{\mathbb{M}}(C)\}) \\
&= \mathbb{M}_{\setminus\varepsilon} \setminus \{m \in \mathbb{M}_{\setminus\varepsilon} \mid \exists m' \in (\mathbb{M}_{\setminus\varepsilon} \setminus M) \cup \{\varepsilon\}.\ (m, m') \in T_{\mathbb{M}}(C)\} \\
&= \{m \in \mathbb{M}_{\setminus\varepsilon} \mid \nexists m' \in (\mathbb{M}_{\setminus\varepsilon} \setminus M) \cup \{\varepsilon\}.\ (m, m') \in T_{\mathbb{M}}(C)\} \\
&= \{m \in \mathbb{M}_{\setminus\varepsilon} \mid \forall m' \in (\mathbb{M}_{\setminus\varepsilon} \setminus M) \cup \{\varepsilon\}.\ (m, m') \notin T_{\mathbb{M}}(C)\} \\
&= \{m \in \mathbb{M}_{\setminus\varepsilon} \mid \forall m' \in \mathbb{M}.\ (m, m') \in T_{\mathbb{M}}(C) \Rightarrow m' \in M\} \\
&= \mathrm{wp}(T_{\mathbb{M}}(C), M)
\end{aligned}$$

□

Lemma 6.2. For a statement C and set of states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$

$$P(C, M) = \mathbb{M}_{\setminus\varepsilon} \setminus N(C, \mathbb{M}_{\setminus\varepsilon} \setminus M)$$

Proof. We proceed by induction on the structure of C. For atomic statements, the result follows from Lemma 6.1. For sequential composition and branch, the result follows directly from the induction hypothesis.

It remains to consider loops C^*. Let A be a fixed point of $\lambda X.P(C, X) \cap M$, i.e., $A = P(C, A) \cap M$. Let $B = \mathbb{M}_{\setminus\varepsilon} \setminus A$,

$$\begin{aligned}
B &= \mathbb{M}_{\setminus\varepsilon} \setminus \big(P(C, \mathbb{M}_{\setminus\varepsilon} \setminus B) \cap M\big) \\
&= \big(\mathbb{M}_{\setminus\varepsilon} \setminus P(C, \mathbb{M}_{\setminus\varepsilon} \setminus B)\big) \cup (\mathbb{M}_{\setminus\varepsilon} \setminus M)
\end{aligned}$$

by structural induction hypothesis

$$= N(C,B) \cup (\mathbb{M}_{\setminus\varepsilon} \setminus M)$$

That is, B is a fixed point of $\lambda Y.N(C,Y) \cup (\mathbb{M}_{\setminus\varepsilon} \setminus M)$.

A similar argument shows that if B is a fixed point of $\lambda Y.N(C,Y) \cup (\mathbb{M}_{\setminus\varepsilon} \setminus M)$, then $\mathbb{M}_{\setminus\varepsilon} \setminus B$ is a fixed point of $\lambda X.P(C,X) \cap M$.

Let $A' = P(C^*,M) = \mathrm{gfp}_{\subseteq}\, \lambda X.P(C,X) \cap M$. This means, $\mathbb{M}_{\setminus\varepsilon} \setminus A'$ is a fixed point of $\lambda Y.N(C,Y) \cup (\mathbb{M}_{\setminus\varepsilon} \setminus M)$. Let $B' = N(C^*, \mathbb{M}_{\setminus\varepsilon} \setminus M) = \mathrm{lfp}_{\subseteq} \lambda Y.N(C,Y) \cup (\mathbb{M}_{\setminus\varepsilon} \setminus M)$; hence $\mathbb{M}_{\setminus\varepsilon} \setminus B'$ is a fixed point of $\lambda X.P(C,X) \cap M$. Since A' is maximal, $A' \supseteq \mathbb{M}_{\setminus\varepsilon} \setminus B'$. Since B' is minimal, $B' \subseteq \mathbb{M}_{\setminus\varepsilon} \setminus A'$, and $A' \subseteq \mathbb{M}_{\setminus\varepsilon} \setminus B'$. Hence, $A' = \mathbb{M}_{\setminus\varepsilon} \setminus B'$, i.e., $P(C^*,M) = \mathbb{M}_{\setminus\varepsilon} \setminus N(C^*, \mathbb{M}_{\setminus\varepsilon} \setminus M)$. □

Lemma 6.3. For a statement C and sets of states $M, V \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$P(C,M) = \mathrm{wp}(T_{\mathbb{M}}(C), M)$$
$$N(C,V) = \text{pre+fail}(T_{\mathbb{M}}(C), V)$$

Proof. It is enough to prove the Lemma for one (e.g., negative) side, then for the other side, it follows from Lemmas 6.1 and 6.2.

For the negative side, the proof proceeds by structural induction. For the atomic statements, sequential composition and branching, it directly follows from the definition of pre+fail.

First, note that from the definition of pre+fail we can derive that,

$$\text{pre+fail}(T_{\mathbb{M}}(C_2) \circ T_{\mathbb{M}}(C_1), V) = \text{pre+fail}(T_{\mathbb{M}}(C_1), \text{pre+fail}(T_{\mathbb{M}}(C_2), V))$$

and therefore, for every i, we can show by induction on i that

$$\text{pre+fail}(T_{\mathbb{M}}(C)^i, V) = \left(\lambda X.\, \text{pre+fail}(T_{\mathbb{M}}(C), X)\right)^i\, V$$

For a loop, C^* and a set of states $V \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$N(C^*, V)$$

by definition

$$= \mathrm{lfp}_{\subseteq} \lambda Y. N(C, Y) \cup V$$

by structural induction hypothesis

$$= \mathrm{lfp}_{\subseteq} \lambda Y. \text{pre+fail}(T_{\mathbb{M}}(C), Y) \cup V$$

by Kleene's Fixed Point Theorem

$$= \bigcup_{i=0}^{\infty} (\lambda Y. \text{pre+fail}(T_{\mathbb{M}}(C), Y) \cup V)^i \varnothing$$
$$= \bigcup_{i=0}^{\infty} (\lambda Y. \text{pre+fail}(T_{\mathbb{M}}(C), Y))^i V$$
$$= \bigcup_{i=0}^{\infty} \text{pre+fail}(T_{\mathbb{M}}(C)^i, V)$$

since pre+fail preserves upper bounds in its first argument

$$= \text{pre+fail}(\bigcup_{i=0}^{\infty} T_{\mathbb{M}}(C)^i, V)$$

by Kleene's Fixed Point Theorem

$$= \text{pre+fail}(\mathrm{lfp}_{\subseteq} \lambda Y. \Delta_{\mathbb{M}} \cup (Y \circ T_{\mathbb{M}}(C)), V)$$
$$= \text{pre+fail}(T_{\mathbb{M}}(C^*), V)$$

□

Lemma 6.4. For the loop C_{loop} in (6.3), s.t.

$$C_{\text{loop}} = ([\psi_{\text{ent}}]\,;\, C_{\text{body}})^*\,;\, [\varphi_{\text{exit}}]$$

and a set of states $M \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$R_\forall \subseteq P(C_{\text{loop}}, M)$$
$$R_\exists \setminus N(C_{\text{loop}}, \mathbb{M}_{\setminus\varepsilon} \setminus M) \subseteq P(C_{\text{loop}}, M)$$

Proof. The result for the existential recurrent set $R_\exists$ follows from Lemma 6.2 independently of the definition of $R_\exists$. We now prove the result for the universal recurrent set.

Let $R = P(C_{\text{loop}}, \varnothing)$, then following the definition of the positive side of a loop,

$$R = \mathrm{gfp}_{\subseteq} \lambda X. \big([\![\neg\psi_{\text{ent}}]\!] \cup P(C_{\text{body}}, X)\big) \cap [\![\neg\varphi_{\text{exit}}]\!]$$

using $[\![\neg\varphi_{\text{exit}}]\!] \cap [\![\neg\psi_{\text{ent}}]\!] = \varnothing$, and Lemma 6.3

$$= \text{gfp}_{\subseteq}\, \lambda X.\, \text{wp}(C_{\text{body}}, X) \cap [\![\neg\varphi_{\text{exit}}]\!]$$

Since P is monotone in its second argument (as wp is), then for every M it holds that $R \subseteq P(C_{\text{loop}}, M)$.

From the definition of a universal recurrent set of a loop $R_\forall$, (6.4),

$$R_\forall \subseteq [\![\neg\varphi_{\text{exit}}]\!] \wedge R_\forall \subseteq \text{wp}(C_{\text{body}}, R_\forall)$$

That is,

$$R_\forall \subseteq \text{wp}(C_{\text{body}}, R_\forall) \cap [\![\neg\varphi_{\text{exit}}]\!]$$

From Tarski's Fixed Point Theorem

$$R_\forall \subseteq \text{gfp}_{\subseteq}\, \lambda X.\, \text{wp}(C_{\text{body}}, X) \cap [\![\neg\varphi_{\text{exit}}]\!]$$

That is, for every $M \subseteq \mathbb{M}_{\setminus\varepsilon}$

$$R_\forall \subseteq R \subseteq P(C_{\text{loop}}, M)$$

□

Theorem 6.1. The alternative characterizations of the positive side of the loop: (6.6) and (6.7) – under-approximate the original characterization (6.5). That is, for $M \subseteq \mathbb{M}_{\setminus\varepsilon}$,

$$P^{\exists}(C_{\text{loop}}, M) \subseteq P(C_{\text{loop}}, M)$$
$$P^{\forall}(C_{\text{loop}}, M) \subseteq P(C_{\text{loop}}, M)$$

Proof. Let

$$C_{\text{loop}} = ([\psi_{\text{ent}}]\,;\, C_{\text{body}})^*\,;\, [\varphi_{\text{exit}}]$$

be a loop as in (6.3). Let $R_\exists$ be its existential recurrent set; P_{rest} be a set of states; $N_{\text{rest}} = \mathbb{M}_{\setminus\varepsilon} \setminus P_{\text{rest}}$; $P_{\text{loop}} = P(C_{\text{loop}}, P_{\text{rest}})$; $N_{\text{loop}} = N(C_{\text{loop}}, N_{\text{rest}}) = \mathbb{M}_{\setminus\varepsilon} \setminus P_{\text{loop}}$. Note that P_{loop} and N_{loop} are the original characterizations of the positive and negative sides as in (6.5).

Then, it holds that

$$[\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}} \subseteq P_{\text{loop}} \tag{6.8}$$

$$R_\exists \setminus N_{\text{loop}} \subseteq P_{\text{loop}} \tag{6.9}$$

$$\text{For } M \subseteq P_{\text{loop}} \text{ we have } \text{wp}(C_{\text{body}}, M) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup ([\![\psi_{\text{ent}}]\!] \cap P_{\text{rest}})) \subseteq P_{\text{loop}} \tag{6.10}$$

Equation (6.8) can be seen from (6.5) and describes the states that immediately cause successful termination of the loop. Equation (6.9) is due to Lemma 6.4. Equation (6.10) is due to the following.

$$P_{\text{loop}} = \text{gfp}_{\subseteq}\, \lambda X.\big([\![\neg\psi_{\text{ent}}]\!] \cup P(C_{\text{body}}, X)\big) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup P_{\text{rest}})$$

Then

$$P_{\text{loop}} = \big([\![\neg\psi_{\text{ent}}]\!] \cup P(C_{\text{body}}, P_{\text{loop}})\big) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup P_{\text{rest}})$$

That is, if $M \subseteq P_{\text{loop}}$ then $([\![\neg\psi_{\text{ent}}]\!] \cup \text{wp}(C_{\text{body}}, M)) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup P_{\text{rest}}) \subseteq P_{\text{loop}}$.

Now, consider the expression

$$([\![\neg\psi_{\text{ent}}]\!] \cup \text{wp}(C_{\text{body}}, M)) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup P_{\text{rest}})$$

Due to $[\![\neg\varphi_{\text{exit}}]\!] \cap [\![\neg\psi_{\text{ent}}]\!] = \varnothing$

$$= ([\![\neg\psi_{\text{ent}}]\!] \cup P_{\text{rest}}) \cup (\text{wp}(C_{\text{body}}, M) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup P_{\text{rest}}))$$

splitting the second occurrence of $P_{\text{rest}} = ([\![\psi_{\text{ent}}]\!] \cap P_{\text{rest}}) \cup ([\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}})$

$$\begin{aligned} &= ([\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}}) \cup \\ &\quad (\text{wp}(C_{\text{body}}, M) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup ([\![\psi_{\text{ent}}]\!] \cap P_{\text{rest}}))) \cup \\ &\quad (\text{wp}(C_{\text{body}}, M) \cap [\![\neg\psi_{\text{ent}}]\!] \cap P_{\text{rest}}) \end{aligned}$$

Note that the first and last disjuncts are already known to be included P_{loop} due to (6.8), and we can just forget about them here. The more important result is that, if $M \subseteq P_{\text{loop}}$, then $\text{wp}(C_{\text{body}}, M) \cap ([\![\neg\varphi_{\text{exit}}]\!] \cup ([\![\psi_{\text{ent}}]\!] \cap P_{\text{rest}})) \subseteq P_{\text{loop}}$. That is, we established (6.10).

Let $P^{\exists}_{\text{loop}} = P^{\exists}(C_{\text{loop}}, P_{\text{rest}})$ Now, by definition (6.6), $P^{\exists}_{\text{loop}}$ is the *smallest* set satisfying (6.8), (6.9), and (6.10) (as it is characterized as a least fixed point of a matching function). Hence, $P^{\exists}_{\text{loop}} \subseteq P_{\text{loop}}$.

For $P^{\forall}_{\text{loop}}$, the proof proceeds in a similar way. For a universal recurrent set $R_{\forall}$ and for the solution of the original equations (6.5), it holds that

$$([\![\varphi_{\text{exit}}]\!] \cap P_{\text{rest}}) \setminus N_{\text{loop}} \subseteq P_{\text{loop}} \tag{6.11}$$

$$R_{\forall} \subseteq P_{\text{loop}} \tag{6.12}$$

$$\text{For } M \subseteq P_{\text{loop}} \text{ we have } ([\![\neg\varphi_{\text{exit}}]\!] \cap \text{pre}(C_{\text{body}}, M)) \setminus N_{\text{loop}} \subseteq P_{\text{loop}} \tag{6.13}$$

Equations (6.11) and (6.13) are due to Lemma 6.2. Equation (6.12) is due to Lemma 6.4. From (6.7), we characterize $P^{\forall}_{\text{loop}}$ to be is the *smallest* set that satisfies (6.11), (6.12), and (6.13). Thus, $P^{\forall}_{\text{loop}} \subseteq P_{\text{loop}}$. □

Theorem 6.2. For a statement C and $p, n \in \mathbb{D}_\sharp$ s.t. $\gamma_\sharp(p) \cap \gamma_\sharp(n) = \varnothing$

$$N^\sharp(C, n) \supseteq N(C, \gamma_\sharp(n))$$
$$P^\flat(C, p, n) \subseteq P(C, \mathbb{M}_{\setminus\varepsilon} \setminus \gamma_\sharp(n))$$

As a result, for a top-level program C_{prg},

$$\gamma_\sharp(N^\sharp(C_{\text{prg}}, \bot_\sharp)) \supseteq N(C_{\text{prg}}, \varnothing)$$

(i.e., it over-approximates input states that may lead to failure), and

$$\gamma_\sharp(P^\flat(C_{\text{prg}}, \top_\sharp, \bot_\sharp)) \subseteq P(C_{\text{prg}}, \mathbb{M}_{\setminus\varepsilon})$$

(i.e., it under-approximates safe input states)

Proof. The result for $N^\sharp$ is a standard result for over-approximate computations.

For $P^\flat$, it follows from the use of subtraction. For a statement C and disjoint $d, n \in \mathbb{D}_\sharp$, $P^\flat(C, d, n)$ is defined as $q - N^\sharp(C, n)$ for some $q \in \mathbb{D}_\sharp$ (for the proof, it does not matter, how q is computed). From the definition of subtraction,

$$\gamma_\sharp(P^\flat(C, d, n)) \subseteq \mathbb{M}_{\setminus\varepsilon} \setminus \gamma_\sharp(N^\sharp(C, n))$$

From the result for $N^\sharp$

$$\gamma_\sharp(P^\flat(C, d, n)) \subseteq \mathbb{M}_{\setminus\varepsilon} \setminus N(C, \gamma_\sharp(n))$$
$$\gamma_\sharp(P^\flat(C, d, n)) \subseteq P(C, \mathbb{M}_{\setminus\varepsilon} \setminus \gamma_\sharp(n))$$

Then, for a top-level program C_{prg},

$$\gamma_\sharp(P^\flat(C, \top_\sharp, \bot_\sharp)) \subseteq P(C_{\text{prg}}, \mathbb{M}_{\setminus\varepsilon})$$

□

Chapter 7

Conclusion

In this work, we presented our take on the problem of finding non-terminating executions in programs, using the framework of abstract interpretation. We started by defining a notion of a program and its trace semantics. We introduced set-of-states abstraction which is a step away from concrete (or, in a sense, *theoretical*) analysis that manipulates sets of traces towards a state-based analysis that can be made computable (or *practical*) by performing further memory abstraction. We introduced a notion of a recurrent set (and we gave two definitions that use different modalities), which is a set-of-states abstraction of a set of non-terminating execution postfixes. Thus, we split the problem of proving non-termination of a program into two sub-problems: finding a recurrent set and showing its reachability. In Chapters 3 and 4 we focused on practical aspects of the former sub-problem and introduced two different analyses that find recurrent sets in programs. The analysis of Chapter 3 was formulated for numeric programs and was based on backward analysis and trace partitioning. The analysis of Chapter 4, while arguably less sophisticated, was based on forward analysis and was suitable to analyse non-numeric (e.g., heap-manipulating) programs. Finally, we demonstrated that recurrent sets are useful not only as a sub-problem of proving non-termination. Chapter 6 is an extensive example of how we can use the notion of recurrent set in a backward safety analysis.

The reader could notice that the concepts that we use in practical chapters (Chapter 4 would be a particularly good example) are somewhat different from the concepts that we use in the theoretical discussions of Chapter 2. In Chapter 2, we talk about program graphs and their executions as sequences of program states. In Chapter 4, we talk

about abstract memory states of a structured program. This is the result of a sequence of abstractions that we build in this work: a structured program represents a program graph (an unstructured program), and a set of memory states of a structured program represents a set of program states at a particular control location; finally, this set of program states is an abstraction of a certain set of program executions. Thus, the manipulations that a practical analysis performs over abstract memory states eventually map to manipulations over sets of executions.

Where do we go from this point? In this work, we focused on a *sub-problem* of proving non-termination, and now, we believe, it is time to look at the *full* problem once again.

First, we should ask ourselves, whether proving non-termination has a practical value and whether it occupies a niche alongside other analyses. So far, we have vaguely identified two potential use cases for it. One is debugging, i.e., finding non-terminating behaviours in programs that are supposed to terminate. It is tempting here to make an analogy with reachability analysis where testing, symbolic execution, bounded model checking and other techniques, which provide incomplete ways to find safety violations, have become as important as verification techniques, which prove absence of those violations.

In this respect, we are (moderately) enthusiastic about the analysis of Chapter 4. The approach described in Chapter 4 is very simplistic, but at the same time extensible. It works (on a very high level) by constructing and analysing an abstract reachability graph, something that many software model-checkers do in one way or another. It would be interesting either to extend the original procedure with a more expressive abstract domain, extrapolation operations, and maybe limited backward analysis, or to re-formulate it within the framework of another existing model-checking procedure, e.g., Impact [McM06], and see whether it can find practical non-termination bugs. There exist extensible tools, like CPAChecker [BHT07; BHT08], that facilitate this kind of integration of different analyses.

Also, we believe that there is value in improving the trace partitioning scheme for the analysis of Chapter 3. One high level point of view on what this analysis does is that it tries to identify the path or paths through the program that are taken by non-terminating executions. We believe this is a powerful idea that requires more research.

Improving admissible forms of paths (path domain), e.g., allowing the paths to be expressed in some form of regular expressions or temporal logic formulas, could significantly improve the precision of the analysis.

Then, if we continue on the path of proving non-termination with abstract interpretation, we face the problem of showing (definite) reachability of our recurrent sets. This is not a new problem. It has been around for a while, in particular, in the form of proving feasibility of abstract counterexamples [Ber+13]. It could be the case that this problem has a good solution within the framework of abstract interpretation. For example, Chapter 6 makes an attempt to build an under-approximating reachability analysis, based on subtraction operation, but this direction still requires more work.

Additionally, so far, our experiments with non-numeric domains were confined to shape analysis with 3-valued logic. While this is a good *example* of a non-numeric domain, it would be more practically interesting to adapt our techniques a to a more popular shape analysis domain, e.g, to separation logic. It would be especially interesting to see, whether and how we can make use of existing techniques specific to separation logic, like bi-abduction [Cal+11].

Finally, most of the discussion in this work was from the point of view that *all* non-terminating behaviours are equally undesired. There are settings though, where this is not the case, and there may exist a notion of an acceptable non-terminating execution. For example, we may want to analyse a reactive system as a whole, and not its dispatch routines separately. Then, an acceptable non-terminating behaviour is that the system alternates infinitely often between accepting a new request and producing a response to it. And a possible undesired non-terminating behaviour is that the system gets stuck in an infinite loop while producing a response and never gets to accept the next request. Another example comes from functional programming, where one can work with functions on infinite objects. A desired property of such a function is often not termination but *productiveness*, i.e., it is acceptable that the function does not terminate for some inputs, but it is desired that in this case, at the limit, it construct a valid infinite object as an output. A natural question is whether non-termination analyses are applicable in settings where not all non-terminating behaviours are undesired, i.e., whether they can find non-terminating behaviours that have specific (undesired) properties.

We anticipate that they can. It seems that in the examples above the distinction between an acceptable and an undesired non-terminating behaviours can be expressed using a combination of safety and weak *fairness*. In the reactive system example, an acceptable non-terminating behaviour alternates between accepting a request and sending a response infinitely often, or in other words:

(i) Every accept is followed by a response and vice versa. This is a safety property.

(ii) In a non-terminating execution, both accepts and responses happen infinitely often. This is weak fairness.

To find execution prefixes where some accept is *not* followed by a response or vice versa, we can use a safety analysis, e.g., an abstract interpreter, an abstraction refinement model checker, etc., but this is out of scope of our work. To find non-terminating behaviours, where an accept or a response happens only a finite number of times (i.e., eventually never happens), we can use a non-termination analysis, in the following way. First, we identify *fair* program statements, that need to happen infinitely often in an acceptable non-terminating execution (respectively, an undesired non-terminating behaviour would eventually never execute some fair statement). In the reactive system example, they would correspond to retrieving a request and sending a response. Then, we run the analysis as follows:

(i) Modify the original program by replacing a single fair statement with a loop exit.

(ii) Run a non-termination analysis.

(iii) If the analysis can prove non-termination of the modified program (e.g., it finds a reachable non-empty recurrent set), then the original program has an undesired non-terminating behaviour.

(iv) Repeat this process for every fair statement.

Bibliography

[Arn+06] Gilad Arnold et al. “Combining Shape Analyses by Intersecting Abstractions”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by E. Allen Emerson and Kedar S. Namjoshi. Vol. 3855. Lecture Notes in Computer Science. Springer, 2006, pp. 33–48.

[ASV12] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. “Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis”. In: *Asian Symposium on Programming Languages and Systems (APLAS)*. Ed. by Ranjit Jhala and Atsushi Igarashi. Vol. 7705. Lecture Notes in Computer Science. Springer, 2012, pp. 157–172.

[Bag+05] Roberto Bagnara et al. “Precise widening operators for convex polyhedra”. In: *Sci. Comput. Program.* 58.1-2 (2005), pp. 28–56.

[BB14] Armin Biere and Roderick Bloem, eds. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014.

[BBP14] Alexey Bakhirkin, Josh Berdine, and Nir Piterman. “Backward Analysis via over-Approximate Abstraction and under-Approximate Subtraction”. In: *Static Analysis Symposium (SAS)*. Ed. by Markus Müller-Olm and Helmut Seidl. Vol. 8723. Lecture Notes in Computer Science. Springer, 2014, pp. 34–50.

[BBP15] Alexey Bakhirkin, Josh Berdine, and Nir Piterman. “A Forward Analysis for Recurrent Sets”. In: *Static Analysis Symposium (SAS)*. Ed. by Sandrine Blazy and Thomas Jensen. Vol. 9291. Lecture Notes in Computer Science. Springer, 2015, pp. 293–311.

[Ber+06] Josh Berdine et al. “Automatic Termination Proofs for Programs with Shape-Shifting Heaps”. In: *Computer-Aided Verification (CAV)*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 386–400.

[Ber+13] Josh Berdine et al. “Resourceful Reachability as HORN-LA”. In: *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*. Ed. by Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. Lecture Notes in Computer Science. Springer, 2013, pp. 137–146.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. “Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis”. In: *Computer-Aided Verification (CAV)*. Ed. by Werner Damm and Holger Hermanns. Vol. 4590. Lecture Notes in Computer Science. Springer, 2007, pp. 504–518.

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. “Program Analysis with Dynamic Precision Adjustment”. In: *Automated Software Engineering (ASE)*. IEEE Computer Society, 2008, pp. 29–38.

[BHZ07] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. “Widening operators for powerset domains”. In: *STTT* 9.3-4 (2007), pp. 413–414.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. “The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems”. In: *Sci. Comput. Program.* 72.1-2 (2008), pp. 3–21.

[BJ06] Thomas Ball and Robert B. Jones, eds. *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[Bou93a] François Bourdoncle. “Abstract Debugging of Higher-Order Imperative Languages”. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Robert Cartwright. ACM, 1993, pp. 46–55.

[Bou93b] François Bourdoncle. “Efficient chaotic iteration strategies with widenings”. In: *Formal Methods in Programming and Their Applications*. Ed. by Dines Bjørner, Manfred Broy, and Igor V. Pottosin. Vol. 735. Lecture Notes in Computer Science. Springer, 1993, pp. 128–141.

[BP16] Alexey Bakhirkin and Nir Piterman. “Finding Recurrent Sets with Backward Analysis and Trace Partitioning”. In: *Tools and Algorithms for the Construction and Analysis of System (TACAS)*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 17–35.

[BPR13] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. “Solving Existentially Quantified Horn Clauses”. In: *Computer-Aided Verification (CAV)*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 869–882.

[Bro+11] Marc Brockschmidt et al. “Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode”. In: *Formal Verification of Object-Oriented Systems (FoVeOOS)*. Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Vol. 7421. Lecture Notes in Computer Science. Springer, 2011, pp. 123–141.

[Bro+16] Marc Brockschmidt et al. “T2: Temporal Property Verification”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference (TACAS)*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 387–393.

[Cal+11] Cristiano Calcagno et al. “Compositional Shape Analysis by Means of Bi-Abduction”. In: *J. ACM* 58.6 (2011), p. 26.

[CC12] Patrick Cousot and Radhia Cousot. “An abstract interpretation framework for termination”. In: *Principles of Programming Languages (POPL)*. Ed. by John Field and Michael Hicks. ACM, 2012, pp. 245–258.

[CC77] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Principles of Programming Languages (POPL)*. Ed.

by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252.

[CC79] Patrick Cousot and Radhia Cousot. “Systematic Design of Program Analysis Frameworks”. In: *Principles of Programming Languages (POPL)*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen. ACM Press, 1979, pp. 269–282.

[CC92] Patrick Cousot and Radhia Cousot. “Abstract Interpretation Frameworks”. In: *J. Log. Comput.* 2.4 (1992), pp. 511–547.

[CC99] Patrick Cousot and Radhia Cousot. “Refining Model Checking by Abstract Interpretation”. In: *Autom. Softw. Eng.* 6.1 (1999), pp. 69–95.

[CH78] Patrick Cousot and Nicolas Halbwachs. “Automatic Discovery of Linear Restraints Among Variables of a Program”. In: *Principles of Programming Languages (POPL)*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 84–96.

[Che+14] Hong Yi Chen et al. “Proving Nontermination via Safety”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. Springer, 2014, pp. 156–171.

[Che64] N. V. Chernikova. “Algorithm for finding a general formula for the non-negative solutions of a system of linear equations”. In: *USSR Computational Mathematics and Mathematical Physics* 4.4 (1964), pp. 151–152.

[Che65] N. V. Chernikova. “Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities”. In: *USSR Computational Mathematics and Mathematical Physics* 5.2 (1965), pp. 228–233.

[Che68] N. V. Chernikova. “Algorithm for discovering the set of all the solutions of a linear programming problem”. In: *USSR Computational Mathematics and Mathematical Physics* 8.6 (1968), pp. 281–293.

[CKP15] Byron Cook, Heidy Khlaaf, and Nir Piterman. “On Automation of CTL* Verification for Infinite-State Systems”. In: *Computer Aided Verification (CAV)*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 13–29.

[Cla77] Edmund M. Clarke. “Program Invariants as Fixed Points (Preliminary Reports)”. In: *Foundations of Computer Science (FOCS)*. IEEE Computer Society, 1977, pp. 18–29.

[Coo+14] Byron Cook et al. “Disproving termination with overapproximation”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2014, pp. 67–74.

[Cos+05] Alexandru Costan et al. “A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs”. In: *Computer Aided Verification (CAV)*. Ed. by Kousha Etessami and Sriram K. Rajamani. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 462–475.

[Cou+05] Patrick Cousot et al. “The ASTREÉ Analyzer”. In: *European Symposium on Programming (ESOP)*. Ed. by Shmuel Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 21–30.

[Cou15] Patrick Cousot. “Sound Verification by Abstract Interpretation”. 2015.

[Cou81] Patrick Cousot. “Semantic foundations of program analysis”. In: *Program Flow Analysis: Theory and Applications*. Ed. by S S Muchnick and N D Jones. Prentice-Hall, 1981, pp. 303–342.

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Termination proofs for systems code”. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006, pp. 415–426.

[CR16] Marsha Chechik and Jean-François Raskin, eds. *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016.

[CYO01] Cristiano Calcagno, Hongseok Yang, and Peter W. O’Hearn. “Computability and Complexity Results for a Spatial Assertion Language for Data Structures”. In: *Asian Workshop on Programming Languages and Systems (APLAS)*. 2001, pp. 289–300.

[Gie+14] Jürgen Giesl et al. “Proving Termination of Programs Automatically with AProVE”. In: *International Joint Conference on Automated Reasoning (IJCAR)*. Ed. by Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Vol. 8562. Lecture Notes in Computer Science. Springer, 2014, pp. 184–191.

[Gup+08] Ashutosh Gupta et al. “Proving non-termination”. In: *Principles of Programming Languages (POPL)*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 147–158.

[Hei+16] Matthias Heizmann et al. “Ultimate Automizer with Two-track Proofs - (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of System (TACAS)*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 950–953.

[HHP14] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. “Termination Analysis by Learning Terminating Programs”. In: *Computer-Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 797–813.

[HMT71] L. Henkin, J. D. Monk, and A. Tarski. *Cylindric Algebras: Part I*. North-Holland, 1971.

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. “Verification of Real-Time Systems using Linear Relation Analysis”. In: *Formal Methods in System Design* 11.2 (1997), pp. 157–185.

[Kle87] Stephen Kleene. *Introduction to Metamathematics*. Second. North-Holland, 1987.

[LA+07] Tal Lev-Ami et al. *Backward Analysis for Inferring Quantified Preconditions*. Tech. rep. TR-2007-12-01. Tel Aviv University, Dec. 2007.

[Lar+14] Daniel Larraz et al. “Proving Non-termination Using Max-SMT”. In: *Computer-Aided Verification (CAV)*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 779–796.

[LH14] Jan Leike and Matthias Heizmann. “Geometric Series as Nontermination Arguments for Linear Lasso Programs”. In: *International Workshop on Termination (WST)*. 2014.

[LMS04] Tal Lev-Ami, Roman Manevich, and Shmuel Sagiv. “TVLA: A system for generating abstract interpreters”. In: *Building the Information Society (IFIP)*. Ed. by René Jacquart. Vol. 156. IFIP. Kluwer/Springer, 2004, pp. 367–375.

[LQC15] Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. “Termination and non-termination specification inference”. In: *Programming Language Design and Implementation (PLDI)*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 489–498.

[McM06] Kenneth L. McMillan. “Lazy Abstraction with Interpolants”. In: *Computer Aided Verification (CAV)*. Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. Lecture Notes in Computer Science. Springer, 2006, pp. 123–136.

[Min06] Antoine Miné. “The octagon abstract domain”. In: *Higher-Order and Symbolic Computation* 19.1 (2006), pp. 31–100.

[Min13] A. Miné. “Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions”. In: *Science of Computer Programming* (2013), p. 33.

[MR05] Laurent Mauborgne and Xavier Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers”. In: *European Symposium on Programming (ESOP)*. Ed. by Shmuel Sagiv. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005, pp. 5–20.

[PC13] Corneliu Popeea and Wei-Ngan Chin. “Dual analysis for proving safety and finding bugs”. In: *Sci. Comput. Program.* 78.4 (2013), pp. 390–411.

[PR04a] Andreas Podelski and Andrey Rybalchenko. “A Complete Method for the Synthesis of Linear Ranking Functions”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 239–251.

[PR04b] Andreas Podelski and Andrey Rybalchenko. “Transition Invariants”. In: *Logic in Computer Science (LICS)*. IEEE Computer Society, 2004, pp. 32–41.

[Ram30] Frank P. Ramsey. “On a Problem of Formal Logic”. In: *Proc. London Math. Soc.* Vol. 30. 1930, pp. 264–285.

[Rey02] John C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Logic in Computer Science (LICS)*. IEEE Computer Society, 2002, pp. 55–74.

[RHS95] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Principles of Programming Languages (POPL)*. Ed. by Ron K. Cytron and Peter Lee. ACM Press, 1995, pp. 49–61.

[RM07] Xavier Rival and Laurent Mauborgne. “The trace partitioning abstract domain”. In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007).

[RSL10] Thomas W. Reps, Mooly Sagiv, and Alexey Loginov. “Finite differencing of logical formulas for static analysis”. In: *ACM Trans. Program. Lang. Syst.* 32.6 (2010).

[RSY04] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. “Symbolic Implementation of the Best Transformer”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by Bernhard Steffen and Giorgio Levi. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004, pp. 252–266.

[Sag05] Shmuel Sagiv, ed. *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Vol. 3444. Lecture Notes in Computer Science. Springer, 2005.

[Sch99] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[SK06] Axel Simon and Andy King. “Widening Polyhedra with Landmarks”. In: *Asian Symposium on Programming Languages and System (APLAS)*. Ed. by Naoki Kobayashi. Vol. 4279. Lecture Notes in Computer Science. Springer, 2006, pp. 166–182.

[SL04] Bernhard Steffen and Giorgio Levi, eds. *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*. Vol. 2937. Lecture Notes in Computer Science. Springer, 2004.

[SRH96] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation". In: *Theor. Comput. Sci.* 167.1&2 (1996), pp. 131–170.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Trans. Program. Lang. Syst.* 24.3 (2002), pp. 217–298.

[Tar55] Alfred Tarski. "A lattice-theoretical fixpoint theorem and its applications." In: *Pacific J. Math.* 5.2 (1955), pp. 285–309.

[Tar72] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160.

[Tur49] Alan Turing. "Checking a Large Routine". In: *Report of a Conference on High Speed Automatic Calculating machines*. 1949, pp. 67–69.

[UM15] Caterina Urban and Antoine Miné. "Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation". In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen. Vol. 8931. Lecture Notes in Computer Science. Springer, 2015, pp. 190–208.

[VR08] Helga Velroyen and Philipp Rümmer. "Non-termination Checking for Imperative Programs". In: *Tests and Proofs*. Ed. by Bernhard Beckert and Reiner Hähnle. Vol. 4966. Lecture Notes in Computer Science. Springer, 2008, pp. 154–170.

[Wei+09] Christoph Weidenbach et al. "SPASS Version 3.5". In: *Automated Deduction (CADE)*. Ed. by Renate A. Schmidt. Vol. 5663. Lecture Notes in Computer Science. Springer, 2009, pp. 140–145.

[Yor+07] Greta Yorsh et al. "Logical characterizations of heap abstractions". In: *ACM Trans. Comput. Log.* 8.1 (2007).