Disjunction on Demand

Dominik Gabi

Master's Thesis

Chair of Programming Methodology
Department of Computer Science
ETH Zurich

http://www.pm.inf.ethz.ch/

Spring 2011

Supervised by:
Dr. Pietro Ferrara
Prof. Dr. Peter Müller

Chair of Programming Methodology
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Trace Partitioning, as presented by Xavier Rival and Laurent Mauborgne in [11], describes an abstract domain according to the abstract interpretation theory. Most existing static analyses based on abstract interpretation adopt a "reachable states" abstraction. Instead, trace partitioning also allows to dynamically keep track of information on how these states are reached. Making it possible to distinguish between states at the same location that were computed by following a different flow of control can lead to a considerable gain in terms of precision.

This report describes the trace partitioning domain implementation and integration into *Sample*, a novel generic static analyzer. *Sample* already contains an analysis based on reachable state semantics and was developed over the last two years by the *Chair of Programming Methodology* at *ETH Zürich*.

Acknowledgments

I would like to thank my family for the countless free meals provided during the writing of this thesis. Furthermore, I would like to thank Prof. Müller for the opportunity to write this thesis in his group. Last but not least, I would like to thank Pietro Ferrara for his supervision and especially for a surprisingly stable interface to *Sample* during the time of the project.

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Goals . . . 9
1.3 Prerequisites and Terminology . . . 10
1.4 Outline . . . 10

2 Abstract Interpretation **11**
2.1 Concrete Semantics . . . 11
2.2 Abstraction . . . 14
2.2.1 Lattices . . . 14
2.2.2 Galois Connections . . . 14
2.2.3 Soundness of the Semantics . . . 15
2.3 Static Analysis . . . 16
2.4 Example Domains . . . 17
2.4.1 Numerical Domains . . . 17
2.4.2 Programming Language Features . . . 17

3 Trace Partitioning **19**
3.1 Refined Semantics . . . 19
3.2 Partitioning Transition Systems . . . 20
3.3 Trace Partitioning Abstract Domain . . . 22
3.4 Static Analysis . . . 24

4 Sample **25**
4.1 Abstract Domain Representation . . . 25
4.1.1 Lattice . . . 25
4.1.2 Values . . . 25
4.1.3 States . . . 26
4.2 Object Oriented Representation . . . 27
4.2.1 Classes, Methods and Statements . . . 27
4.2.2 Control Flow Graph . . . 27
4.2.3 Control Flow Graph Execution . . . 28
4.2.4 Analysis . . . 28
4.3 Limitations . . . 29

5 Trace Partitioning in Sample **31**
5.1 Domain Representation . . . 31
5.1.1 Tokens . . . 31
5.1.2 Directives . . . 32
5.2 Architecture . . . 32
5.2.1 Partitioned State . . . 33
5.2.2 Partitioning . . . 34

5.3 Integration . . . 36
5.3.1 Core Modification . . . 36
5.3.2 Analysis Interaction . . . 37
5.4 Directives . . . 37
5.4.1 PartitionIf . . . 38
5.4.2 Merge . . . 38
5.4.3 PartitionValue . . . 39
5.4.4 PartitionWhile . . . 41
5.5 User Interface . . . 44
5.5.1 Analysis Setup . . . 44
5.5.2 Adding Directives . . . 44
5.5.3 Running the Analysis . . . 45
5.6 Limitations . . . 45

6 Evaluation **47**
6.1 Partitioning a Conditional . . . 48
6.2 Partitioning over a Variable . . . 49
6.3 Partitioning a Loop . . . 50
6.4 Performance . . . 51

7 Future Work and Conclusion **55**
7.1 Open Questions . . . 55
7.1.1 Creating Directives . . . 55
7.1.2 Integrating Directives . . . 56
7.2 Possible Extensions . . . 56
7.2.1 Heuristics . . . 56
7.2.2 New Directives . . . 56
7.2.3 Domain Specific Directives . . . 56
7.3 Conclusion . . . 56
7.3.1 Shortcomings . . . 57
7.3.2 Experience . . . 57
7.3.3 Contribution . . . 57

Chapter 1

Introduction

Change is the only constant.[1] Unlike in other fields, no deep knowledge of the computing industry is required to acknowledge what the Greek philosophers knew long before the first computer was ever invented. It is doubtful though, that the Greeks ever imagined the pace at which change has happened over the past seventy years.

1.1 Motivation

One of the big changes that has happened during this time is a steady increase in complexity, both in hardware and in software. During the sixties of the last century the idea was introduced that programs can be described and reasoned about using mathematical models [8, 9]. This insight has lead to the development of a wide variation of formal methods that have been applied with varying degrees of success.

In hardware design *Model Checking* has become a standard procedure. This is understandable given the cost of possible errors. The situation in software is a different one. While having a long tradition in compiler design and safety critical projects, formal methods generally have a hard stand. Although *Static Analysis* is performed during every compilation of a program, the properties analyzed are usually limited to rather simple properties such as type checks. Looking at common development environments, there is clearly a gap between what is being developed in academia and what is used in day to day life in the industry.

However, there seems to be an ongoing shift in opinion. Relying ever more on distributed services using the internet has lead to a fragile environment whose exploits are getting more and more attention from the mainstream media. This makes correctness in software no longer a luxury that can only be afforded by military and banking institutions but a necessity for an increasing number of businesses.

This is obviously good news for everyone involved in research, but also for everyone who has – as I have plenty of times – ever struggled with a bug that could have easily been detected by an automatic tool.

1.2 Goals

Abstract Interpretation is a framework describing in a very general way how to soundly approximate mathematical models. *Sample* is a tool for static analysis based on abstract interpretation that is currently being developed by the *Chair of Programming Methodology* at *ETH Zürich*. The goal of this thesis is to extend *Sample* with *Trace Partitioning*, a mechanism within abstract interpretation supplying the analysis with information about the history of control flow, thereby tracking disjunctive information and significantly improving overall precision.

[1] Heraclitus (ca. 535 BC - 475 BC)

1.3 Prerequisites and Terminology

The programming language used in *Sample*, in this project as well as in the code listings of this report, is `Scala`. Generally, apart from the pattern matching mechanism often used in functional programming, I try to use as few language specific features as possible and the code samples should be fairly easy to understand without any prior knowledge of `Scala`. Nonetheless, should the need arise, the reader is referred to [14] for further information.

Diagrams depicting software elements are in *UML*, for which the Wikipedia article provides a good overview at [15]. Since there does not seem to be a standard on how to use *UML* with `Scala`, I use the following conventions:

- Traits are indicated using the stereotype notation «`trait`».
- Case classes (used in pattern matching) are indicated using the stereotype «`case`».
- Type parameters used in inheritance hierarchies are also indicated using the stereotype notation on the arrow («`D`»).

1.4 Outline

The rest of this thesis is organized as follows. Chapters 2 and 3 will present the necessary theoretical background for this thesis. The former discusses abstract interpretation in general whereas the latter focuses on the trace partitioning mechanism. Chapter 4 presents an overview of *Sample*. The implementation of the trace partitioning mechanism is presented in Chapter 5, followed by examples demonstrating its application and a short evaluation in Chapter 6. Finally, Chapter 7 discusses the remaining open questions, makes suggestions for possible future extensions and concludes.

Chapter 2

Abstract Interpretation

This chapter presents the fundamental theoretical background to understand the trace partitioning abstract domain presented in Chapter 3. I will start by formally describing abstract interpretation, continue with its application to static analysis and discuss a few abstract domains.

Abstract Interpretation is a framework that describes the sound approximation of mathematical models. It was initially presented in 1977 by Patrick and Radhia Cousot in [4] though there exist easier texts on the subject. The papers [2] and [5] deserve a special mentioning. The former provides a reader-friendly introduction whereas the latter is a dense explanation of the subject, including all the necessary proofs, that is more suitable as reference work. Furthermore, the already mentioned paper by Mauborgne et al. [11] from which some of the following definitions are taken also provides a very thorough introduction. I will follow along their explanation, though in a less rigorous fashion, focussing less on the formal aspects and more on the intuition while at the same time introducing the used notational conventions. Since the goal of this section is to finally provide insight into trace partitioning I will also borrow their running example.

2.1 Concrete Semantics

A mathematical model for a program P is necessary, in order to talk about its concrete semantics. This is commonly given as a transition system.

Definition 2.1 (Transition System). *A transition system is a tuple* $P = (\Sigma, \Sigma_0, \rightarrow)$*, where*

- Σ *denotes the set of states,*
- $\Sigma_0 \subseteq \Sigma$ *the set of initial states and*
- $\rightarrow \subseteq \Sigma \times \Sigma$ *is the transition relation.*

Example 2.1 (Transition System). To illustrate how to represent a program as a transition system consider the method `ifExample` from Listing 2.1 and its corresponding control flow graph (CFG) in Figure 2.1. The method takes as an argument an integer `x`, sets the `sign` variable depending on whether `x` is smaller than `0` to `-1` or `1` and finally returns `x` divided by `sign`.

The mathematical model for the current discussion and, unless stated otherwise, for the rest of this report assumes that there are variables described by some set X and possible values from another set V. A memory state (also called store) $m \in M$ of the system can be described by a mapping of variables to values, that is $M = X \rightarrow V$. The full state of the system can then be described by mapping each program location (also called control state) $l \in L$ to a memory state, that is $\Sigma = L \times M$.

The transition system for the `ifExample` method is fully determined by the set of initial states Σ_0. Figure 2.2 depicts the system having only two initial states. In the first initial state `x` is set to `-2`, in the second state `x` is initialized with `0`. The state changes after executing each statement of the program.

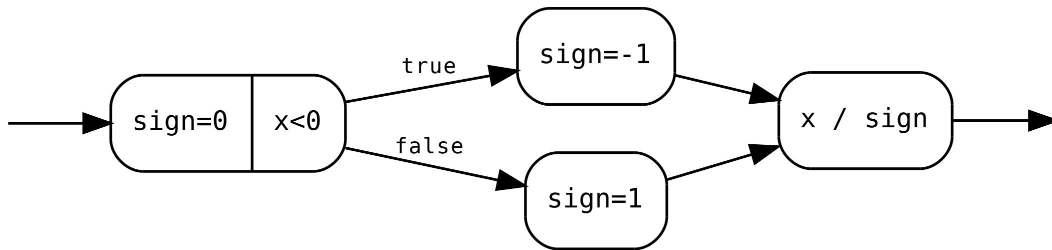

Figure 2.1: The Control Flow Graph for the `ifExample` Method

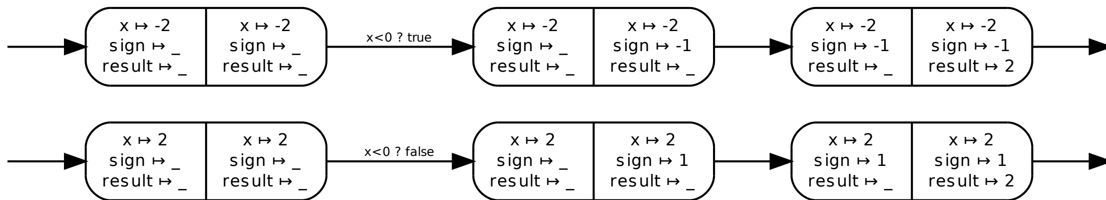

Figure 2.2: The Transition System

Since the blocks of the control flow graph in this particular example consist of single statements, there are two states in the transition system for each block. For the sake of clarity, the edges connecting these two states have not been depicted. The first state represents the input of the block and the second one the output. The input state is the output of the preceding block. If the edge is weighted, indicating that it is the result of a conditional, the state is changed by assuming either the condition (`true`) or its negation (`false`) respectively. ○

```
1  def ifExample(x: Int): Int = {
2      var sign = 0
3      if (x < 0) {
4          sign = -1
5      } else {
6          sign = 1
7      }
8      var y = x / sign
9      y
10 }
```

Listing 2.1: The `ifExample` Method

There are several possible ways to define the semantics $[\![P]\!]$ of the program P. A hierarchical view of several possible definitions is given in [3]. The two relevant definitions for this presentation are the trace semantics and the collecting semantics.

Definition 2.2 (Trace). *A trace is a finite sequence of states* $\sigma = \langle \sigma_0, \sigma_1, ..., \sigma_n \rangle$. *The set of possible traces over the states* Σ *is denoted by* $\Sigma^\star$.

With the definition of traces it is straightforward to define the first type of semantics, namely the trace semantics of a program P.

Definition 2.3 (Finite Partial Trace Semantics). *The finite partial trace semantics describes the set of*

traces that are determined by the transition system P.

$$[\![P]\!] = \{\langle \sigma_0, \sigma_1, ..., \sigma_n \rangle \in \Sigma^\star \mid \sigma_0 \in \Sigma_0 \wedge \forall i, \sigma_i \rightarrow \sigma_{i+1}\}. \tag{2.1}$$

Definition 2.4 (Trace Semantics as Fixed Point). *Alternatively, the trace semantics can be defined recursively as the least fixed point*[1]

$$[\![P]\!] = \mathbf{lfp}^{\subseteq}_{\Sigma_0} F_T \tag{2.2}$$

of the semantic function F_T defined as

$$\begin{aligned} F_T &: \Sigma^\star \rightarrow \Sigma^\star \\ S &\rightarrow S \cup \{\langle \sigma_0, ..., \sigma_n, \sigma_{n+1} \rangle \mid \langle \sigma_0, ..., \sigma_n \rangle \in S \wedge \sigma_n \rightarrow \sigma_{n+1}\}. \end{aligned} \tag{2.3}$$

At each step of the iteration, all current traces in $[\![P]\!]$ are extended by all possible next states as defined by the transition relation. The resulting traces are then added to the set of current traces and the iteration starts over again. Starting with the set Σ_0 of initial states, this will enumerate all possible traces described by P. Note that $[\![P]\!]$ is not necessarily finite and the existence of a fixed point can therefore not be guaranteed.

Given equation 2.1, we can define the semantics collecting the set of all reachable states as follows.

Definition 2.5 (Collecting Semantics). *The states that are reachable in a given transition system are called collecting semantics*

$$[\![P]\!]_C = \{\sigma_n \mid \langle \sigma_0, ..., \sigma_n \rangle \in [\![P]\!]\}. \tag{2.4}$$

Definition 2.6 (Collecting Semantics as Fixed Point). *Analogously to the trace semantics, the collecting semantics can be written in fixed point form*

$$[\![P]\!]_C = \mathbf{lfp}^{\subseteq}_{\Sigma_0} F_C \tag{2.5}$$

using the semantic function F_C defined as

$$\begin{aligned} F_C &: \Sigma \rightarrow \Sigma \\ S &\rightarrow S \cup \{\sigma_{n+1} \mid \sigma_n \in S \wedge \sigma_n \rightarrow \sigma_{n+1}\}. \end{aligned} \tag{2.6}$$

The difference to the iteration computing the trace semantics is that, instead of keeping track of traces, the iteration only looks at single states. In every step, all next states of all current states, as described by the transition relation, are added to the set.

Example 2.2 (Semantics). The concepts presented so far can be illustrated looking at the transition system introduced by Figure 2.2. Each trace of the system starts with an initial state from Σ_0. The two possible initial states are the states on the left side of the figure. The trace semantics contains all possible traces described by the system. To compute the trace semantics as a fixed point, we start with the two traces containing only the initial states. For each trace we then recursively add all possible nodes that are connected to the last state of a trace in the current set of traces. For this example this is a trivial task but note that this is merely due to the restrictive choice of Σ_0. Computing the collecting semantics is trivial as well, since all depicted states are obviously reachable from an initial state, they are all part of $[\![P]\!]_C$. The fixed point computation follows along the same lines as the computation of the trace semantics.

Note that in the general case, the semantics might not be computable. The set of initial states might be infinite, considering all possible values of x. Or it might be prohibitively large, for example, when looking at all possible 32-bit integers for x.

○

Since $[\![P]\!]_C$ in general is undecidable, it is very common to compute an over-approximation of this set and check whether some safety property holds in all states. This is already a form of abstraction, a concept which will be formalized in the next section.

[1] The notion $\mathbf{lfp}^{\subseteq}_{S_0} F$ represents the recursive application of F starting with S_0 until $S_i = S_{i+1}$.

2.2 Abstraction

A single state $\sigma \in \Sigma$ in the mathematical model might be as simple as containing just a mapping from variables to values or it might be as complicated as the actual state of some hardware component. Whichever way, its complexity might be hindering in both formulating as well as verifying interesting properties.

Abstraction removes complexity by focusing on the important aspects of system. This results in two problem domains. The concrete domain of the mathematical structure in question and its simplification, called the abstract domain. Abstract interpretation addresses the problem of relating these two domains in a sound way. That is, how is it possible to guarantee that statements made about an abstract state allow sound conclusions about its concrete counterpart?

2.2.1 Lattices

A part of the solution is provided by the restrictions put on the domains. It is necessary that these are complete lattices.

Definition 2.7 (Lattice). *A partially ordered set* $(S, \sqsubseteq)$ *is a lattice if for any two elements* $s_0, s_1 \in S$

- *there exists a unique* least upper bound $s_u \in S$ *such that* $s_0 \sqsubseteq s_u \wedge s_1 \sqsubseteq s_u$ *denoted by* $s_0 \sqcup s_1$,
- *there exists a unique* greatest lower bound $s_l \in S$ *such that* $s_l \sqsubseteq s_0 \wedge s_l \sqsubseteq s_1$ *denoted by* $s_0 \sqcap s_1$.

A lattice is said to be complete if both the least upper bound and the greatest lower bound are defined for any subset of S.

Definition 2.7 implies that there exists a single element that is the lower bound of all other elements in S called bottom and denoted by $\bot$. Correspondingly, the single element that is the upper bound of all other elements is called top and denoted by $\top$.

Example 2.3 (Complete Lattice). A typical example of complete lattices are type hierarchies like the one depicted in Figure 2.3. It shows a system with four types A, B, C and D where both C and D are subtypes of B. The top and bottom elements have their counterparts in most modern languages. In `Scala`, for example, the top element of the type hierarchy is represented by `Any` and the bottom element by the type `Nothing`. The ordering relation then corresponds to the "subtype", the least upper bound to the "least common supertype" and the greatest lower bound to the "greatest common subtype" relationships. ○

To get an intuition about the function of these lattices it helps to think of their elements in terms of the information they represent. Taking the least upper bound of two elements corresponds to finding the minimum element that encompasses the information of both its arguments and results in a loss of precision. Taking the greatest lower bound means looking for the element representing the information both elements have in common. The top element then represents the information stored in all elements of the lattice together without discerning between single possibilities, which in effect amounts to knowing nothing. On the other end of the lattice, the bottom element represents the conjunction of all elements and, assuming we discern between more than a single state, amounts to a contradiction.

2.2.2 Galois Connections

Knowing the structure of the two domains, it is time to look at functions connecting a concrete domain Σ and its associated abstract domain D. The abstraction maps concrete elements to their abstract counterparts and is usually denoted by $\alpha : \Sigma \to D$. The function mapping abstract elements to the concrete states they describe is called concretization and, by convention, is denoted by $\gamma : D \to \Sigma$.

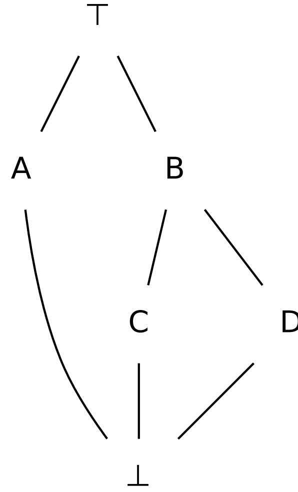

Figure 2.3: A Type Lattice

Definition 2.8 (Galois Connection). *The relation of the two domains is a sound abstraction if these two functions form a Galois connection* $(\Sigma, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D, \sqsubseteq)$ *, that is:*

1. *α and γ are monotone: $\forall \sigma,\ \sigma' \in \Sigma,\ \sigma \subseteq \sigma' \implies \alpha(\sigma) \sqsubseteq \alpha(\sigma')$ and vice versa.*
2. *$\alpha \circ \gamma$ is reductive: $\forall d \in D,\ \alpha \circ \gamma(d) \sqsubseteq d$.*
3. *$\gamma \circ \alpha$ is extensive: $\forall \sigma \in \Sigma,\ \sigma \subseteq \gamma \circ \alpha(\sigma)$.*

Once more, it helps to think in terms of information to develop the intuition about this formalization of sound abstractions. The first point ensures that a less precise element in the concrete domain results in a less precise abstraction and conversely a loss in precision in the abstract leads to a loss of precision in the concrete domain. The last two points are closely related and formalize that while the abstract counterpart of some element may describe more than just the original element, the converse relation does not hold for abstract elements. That is, by means of concretization and subsequent abstraction it is not possible to gain information in the abstract domain.

Example 2.4 (Galois Connection). To further illustrate the points just made, consider the concrete domain of a program describing a set of variables and the abstract domain provided by the type system depicted in Figure 2.3 from the previous example. The monotonicity of abstraction ensures that given two variables `x` of type `C` and `y` of type `D`, the abstraction of their combination given by type `B` must be a supertype of `C` and `D`. Assuming that `x` and `y` are the only variables in the system, monotonicity on the concretization ensures that all variables of type `C` (that is `x`) are a subset of all variables of type `B` (`x` and `y`).

To illustrate the necessity of a reductive $\alpha \circ \gamma$, assume that this restriction is violated. Then starting with `x`, applying the abstraction to get type `C` and continuing with the concretization of `C` would somehow result in a set that does not include `x`. This clearly does not represent the common intuition of a sound abstraction. Arguing for the necessity of the third property can be done along the same lines. ○

2.2.3 Soundness of the Semantics

Since the full transition system of a program is usually not tractable, having related the static aspects of the system, it remains unclear what happens during the transitions of the system. Assuming that

transitions in the concrete system follow a set of rules, corresponding rules for transitions in the abstract domain need to be defined. How these are defined depends on the abstract properties of interest. However, the abstract transitions need to fulfill certain restrictions in order to guarantee soundness.

Definition 2.9 (Soundness of Abstract Operations). *Given a concrete transition rule* $[\![s]\!]_\Sigma : \wp(\Sigma) \to \wp(\Sigma)$ *corresponding to a programs transition system, its abstract counterpart* $[\![s]\!]_D : D \to D$ *preserves soundness if*

$$\forall \sigma \in \wp(\Sigma),\ \alpha([\![s]\!]_\Sigma(\sigma)) \sqsubseteq [\![s]\!]_D(\alpha(\sigma)). \tag{2.7}$$

Executing an abstract operation in the abstract domain results in a new state which describes at least the states that resulted from the execution of the corresponding concrete operation in the concrete domain.

2.3 Static Analysis

Static analysis with abstract interpretation is based on the fixed point computation of the collecting semantics (cf. Definition 2.6). However, instead of actually computing the semantics which might not be decidable, the iteration happens in the abstract domain. Starting with the abstract states representing the initial states of the concrete transition system, each transition of the concrete system is simulated with abstract operations until the set of abstract reachable states converges.

This leads to two problems. First of all, what guarantees that a fixed point computed in the concrete domain corresponds to the fixed point computed in the abstract domain? This question is answered by the so called fixed point transfer theorem as described in [11].

Theorem 2.1 (Fixed Point Transfer). *Given two functions* $F_\Sigma : \Sigma \to \Sigma$ *and* $F_D : D \to D$ *then*

$$\forall \sigma \in \Sigma,\ d \in D,\ \sigma \subseteq \gamma(d) \wedge F_\Sigma \circ \gamma \subseteq \gamma \circ F_D \implies \mathbf{lfp}^{\subseteq}_{\sigma} F_\Sigma \subseteq \mathbf{lfp}^{\subseteq}_{d} F_D \tag{2.8}$$

The premises for this theorem have already been established with the definition of the Galois connection.

The second problem is that of convergence. It is obvious that for some programs, the domain could be of infinite height and hence the fixed point computation may not converge. This problem is addressed using the widening operator instead of the least upper bound, defined as follows:

Definition 2.10 (Widening). *The widening is a binary operator* ∇ *in* D *satisfying*

1. $\forall d, d' \in D, d \sqsubseteq d \nabla d' \wedge d' \sqsubseteq d \nabla d'$
2. *For any sequence* $(d_n)_{n \in \mathbb{N}}$*, the recursive application of the widening operator to the elements of the sequence starting with some* $d_0 \in D$ *will eventually converge.*

Replacing the union used in the fixed point iteration with an operator satisfying the above definition ensures convergence. The widening operator is specific to the abstract domain.

Example 2.5 (Static Analysis). The following example will show a step by step static analysis using the principles presented so far. The basis for this example is again the method `ifExample`, shown in Listing 2.1, but this time with no restrictions on the initial memory state, that is $\Sigma_0 = \{l_0\} \times M$ where l_0 denotes the first program location.

The abstract domain of interest is the sign domain depicted in Figure 2.4 that tracks whether a variable is positive, negative or zero. I will not formally define the abstract operations here, they, however, follow common sense. For example, a negative number multiplied by a negative number results in a positive number. The addition of a negative and a positive number could result in either one and the analysis therefore concludes the result to be $\top$.

Figure 2.5 outlines some of the basic stages of the analysis. ① shows the initial state before the first block as well as its successor state, added by a single iteration of F_R. The initial state is set to $\top$,

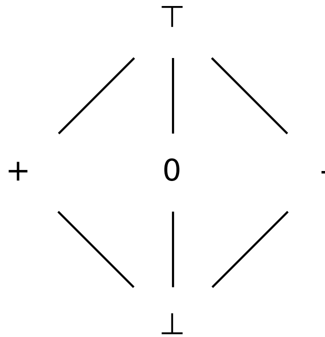

Figure 2.4: The Sign Lattice

meaning that nothing about the environment is assumed. After simulating the execution of the first block using the abstract operations, sign is guaranteed to be 0, therefore its sign must also be 0.

② shows a few steps further in the analysis where the two branch states are already added to the set of known reachable states. A notable difference between the two branches is that when taking the `false` branch, `x` could either be 0 or + and it must therefore be set to $\top$.

③ shows the final state of the analysis. The initial state of the last block must combine the results of the two branches by computing the widening. This has the unfortunate consequence that the resulting state is pretty useless since the little knowledge gained about the sign variable during the computation of the branch is lost.

Convergence is reached in a single iteration over the transition system since the sign domain is really simple and the program does not contain any loops. ○

2.4 Example Domains

Having already hinted at the versatility of the abstract interpretation framework I am now going to provide a quick overview of some of the more commonly used abstract domains.

2.4.1 Numerical Domains

There are tons of domains addressing numerical issues. Although probably not that useful, but often used as an introductory example, the already presented sign domain is one of them. Another, more useful numerical domain is the interval domain which represents the value of a variable by a lower and an upper bound.

The domains seen so far all represent values of single variables and are called non-relational domains. In contrast, relational domains try to, as their name already suggests, connect different variables. The prime example of this type of domain are the polyhedra described in [6]. They infer linear dependences between variables. The Octagons are another example of numerical domains. They track invariants of the from $\pm x \pm y \leqslant c$ and can be implemented efficiently as described in [12].

2.4.2 Programming Language Features

The applicability of abstract interpretation is not limited to numerical domains. For example, most modern object oriented languages use some kind of a heap structure where objects reside in memory. Abstract interpretation can be used to argue about that structure, answering, for example, questions about aliasing.

Furthermore, there are plenty of concepts that can be represented using an abstract domain lattice. To list just a few:

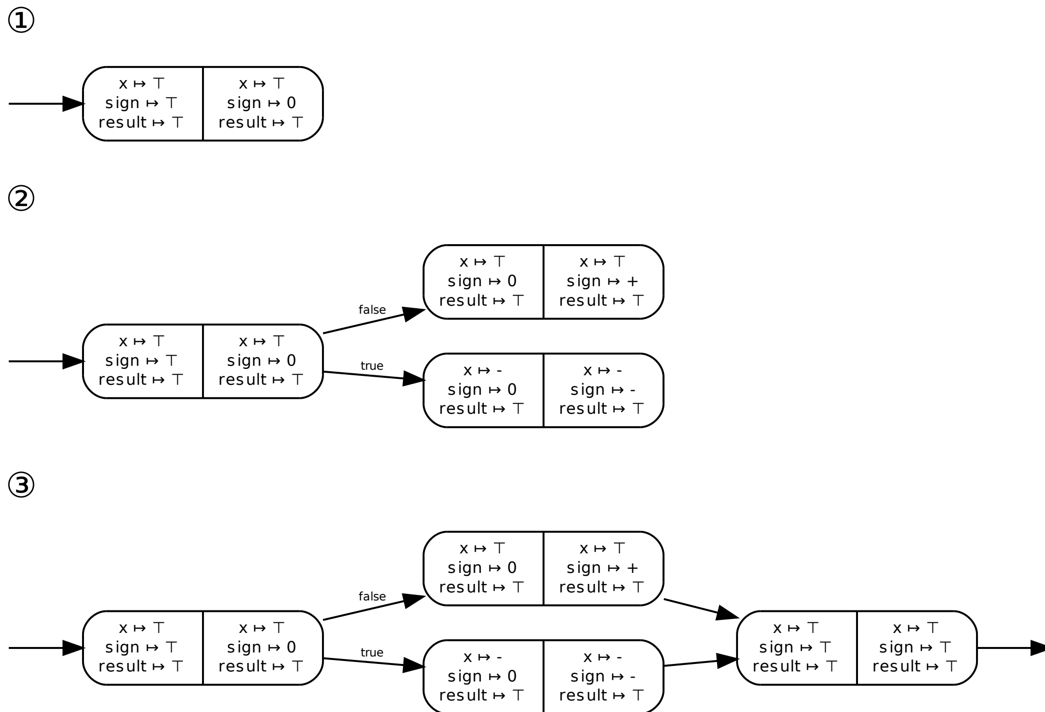

Figure 2.5: Step by Step Static Analysis

- The type system, using a lattice like the one depicted in Figure 2.3.
- Array indices, to prove the safety of array access operations.
- Information about string values [1].
- Exhaustiveness of pattern matching for functional languages [7].

These are just a few of the many possible domains, a search on Google-Scholar for "abstract domain" provides plenty of reading material for the interested reader.

Chapter 3

Trace Partitioning

This chapter describes the trace partitioning abstract domain as it is presented in [13, 11] in detail. The theory is necessary in order to understand the implementation described in Chapter 5. Once more, the focus of the discussion lies on intuition rather than on rigorous formal definitions. All relevant proofs and further examples can be found in the referenced paper.

Static analysis with abstract interpretation as presented in Section 2.3 has proven to be both flexible and efficient. There are, however, cases where the approximation of reachable states provided by the fixed point computation is too coarse to produce meaningful results. This is the case when the proof of a property relies, for example, on the way a state is reached, a piece of information that is completely discarded during the standard analysis.

A possible remedy would be to analyze a more precise approximation of the concrete semantics. Unfortunately, simply abandoning the reachable state semantics in favor of a more precise abstraction (e.g. the trace semantics) has so far turned out to come at too high a price in terms of complexity.

Trace partitioning is an attempt at finding the middle ground between the prohibitive complexity of discerning traces and the overly simplistic view of the reachable state semantics. It does so by effectively partitioning reachable states based on *some* decisions made along the control flow. The theory is very general and fits well within the abstract interpretation framework and can be formalized as an abstract domain.

3.1 Refined Semantics

Before talking about how the partitioning works, it is important to know what exactly a partitioning is and how it can be used to refine the collecting semantics.

Definition 3.1 (Covering and Partition). *A mapping $\delta : E \to \wp(S)$ is said to be a covering of S if*

$$S = \bigcup_{e \in E} \delta(e). \tag{3.1}$$

If additionally all elements of E produce disjunct images in S, i.e.

$$\forall e, e' \in E,\ e \neq e' \wedge \delta(e) \cap \delta(e') = \varnothing, \tag{3.2}$$

δ is called a partitioning of S.

The name trace partitioning is misleading since the theory does not depend on partitions but is sound with coverings as well. Nonetheless, for the sake of simplicity, the further discussion will distinguish between the two only when necessary.

The underlying idea of trace partitioning is to refine the collecting semantics using partitions. To illustrate how, it helps to take another look at the whole abstraction process from the concrete

semantics of a program to the abstract state. Since Galois connections, as defined by Cousot & Cousot, are composable, the abstraction can be split into two parts

$$[\![P]\!] \underset{\alpha_C}{\overset{\gamma_C}{\leftrightarrows}} [\![P]\!]_C \underset{\alpha_D}{\overset{\gamma_D}{\leftrightarrows}} D. \tag{3.3}$$

The two steps are

- the abstraction of the concrete semantics $[\![P]\!]$ to the collecting semantics $[\![P]\!]_C$ followed by
- the abstraction of the reachable states of the collecting semantics to some other abstract domain D.

The first abstraction can be extended to include a partitioning $\delta : E \to [\![P]\!]$. The two steps can then be rewritten as

$$[\![P]\!] \underset{\alpha_\delta}{\overset{\gamma_\delta}{\leftrightarrows}} (E \to [\![P]\!]) \underset{\alpha_D}{\overset{\gamma_D}{\leftrightarrows}} D. \tag{3.4}$$

Here, α_δ describes the abstraction that transforms the concrete semantics into a function that maps elements of some label set E to traces of $[\![P]\!]$. The abstraction is called partitioning abstraction and can be shown to form a Galois connection, provided that δ is in fact a covering or a partitioning. A more formal definition of the abstraction and concretization functions follows.

Definition 3.2 (Partitioning Abstraction). *The partitioning abstraction is defined as*

$$\begin{aligned} \alpha_\delta : [\![P]\!] &\to (E \to [\![P]\!]) \\ \sigma &\mapsto \lambda(e) \cdot \sigma \cap \delta(e) \end{aligned} \tag{3.5}$$

and its corresponding concretization as

$$\begin{aligned} \gamma_\delta : (E \to [\![P]\!]) &\to [\![P]\!] \\ \phi &\mapsto \bigcup_{e \in E} \phi(e). \end{aligned} \tag{3.6}$$

Note that so far no decision about the form of δ has been made. This leaves a large degree of freedom in designing the abstract domain. Consider, on the one hand, the partitioning that maps a unique label to each trace of $[\![P]\!]$. This amounts to having access to the full trace semantics during the further analysis at the expense of having to deal with the accompanying complexity. On the other hand, a partitioning that collects traces ending in the same state results in the classical situation of dealing with the collecting semantics during the analysis. These are the two extremes, anywhere in between is possible. As always there is a trade-off between complexity and precision of the analysis. The fundamental difference to other approaches to this problem is that this trade-off can be managed with great flexibility using the mechanism introduced by a custom partitioning function.

This extension is the foundation of trace partitioning. The rest of this chapter will be concerned with defining an appropriate partition function as well as constructing the lattice structure required for a sound abstract domain.

3.2 Partitioning Transition Systems

The extension of the traditional abstraction and the choice of the partitioning in particular leave many questions to be answered. The goal of this section is to define a useful partitioning as well as an ordering on partitions that can be used to define an abstract domain. Furthermore, it is important to introduce the notion of semantic adequacy, showing that this extension describes the same program as the original semantics.

The underlying structure to partition is the program whose semantics is represented by the transition system. But what does it mean to partition such a system? This is probably the most complex aspect of trace partitioning. It requires an extension of the notion of transition systems presented in Definition 2.1.

Definition 3.3 (Partitioned System, Trivial Extension). *A partitioned transition system P^T is an extension of a transition system $P = (\Sigma, \Sigma_0, \rightarrow)$ represented as a tuple $(T, \Sigma^T, \Sigma_0^T, \rightarrow^T)$ where*

- *T denotes a set of tokens (or labels),*
- *$\Sigma^T = T \times \Sigma$ denotes the set of states,*
- *$\Sigma_0^T \subseteq T \times \Sigma_0$ the set of initial states and*
- *$\rightarrow^T \subseteq \Sigma^T \times \Sigma^T$ is the transition relation.*

Furthermore, the trivial extension of P is defined as the partitioned system with a single token t (i.e. $T = \{t\}$) where all states are extended with t and the extended transition relation completely ignores the newly introduced token.

The only difference to the original transition system is that every state now comes with an additional token of some token set T. The token set T can be thought of as a set of available labels that can be associated with states. This makes it possible to assign tokens to whole traces of P, effectively providing a way to define the partitioning δ.

Another way of looking at the tokens is interpreting them as an extension of the control state as it is done in the presentation of Mauborgne and Rival. The extended control state is then defined as $L^T = T \times L$, a notion which I too will use in the further presentation.

Definition 3.4 (Partitioned Semantics). *The partitioned semantics $[\![P^T]\!]_P$ of P^T is described by applying the partitioning abstraction (Definition 3.2) to the concrete semantics using the partitioning*

$$\begin{aligned} \delta : [\![P]\!] &\rightarrow (L \rightarrow [\![P]\!]) \\ S &\mapsto \lambda(l) \cdot \{s \in S \,|\, \exists \sigma \in \Sigma^T, s = \langle ..., (l, \sigma) \rangle\}. \end{aligned} \tag{3.7}$$

In order to relate two partitioned systems P^T and $P^{T'}$ that are based on the same original transition system P, a function $\tau : T \rightarrow T'$ relating the labels is required. This function is called forget function since it is mainly used to relate a more complicated set of labels to a simpler one by systematically "forgetting" information. The function is trivially extended to states, traces, and sets of traces by applying τ to the associated token, all occurring states, and all occurring traces of the set respectively.

Definition 3.5 (Coverings, Partitions, Completeness). *For a transition system P and its two extensions P^T and $P^{T'}$:*

1. *P^T is a τ-covering of $P^{T'}$ if for every transition in P^T there exists a corresponding transition in $P^{T'}$, that is*
   - $\Sigma_0^{T'} \subseteq \Sigma_0^T$
   - $\forall \sigma_0 \in \Sigma^T,\ \sigma_1' \in \Sigma^{T'},\ \tau(\sigma_0) \rightarrow^{T'} \sigma_1' \implies \exists \sigma_1 \in \Sigma^T,\ \tau(\sigma_1) = \sigma_1' \wedge \sigma_0 \rightarrow^T \sigma_1.$
2. *P^T is a τ-partition of $P^{T'}$ if additionally the corresponding transition is unique*
   - $\forall \sigma' \in \Sigma_0^{T'},\ \exists! \sigma \in \Sigma_0^T,\ \sigma' = \tau(\sigma)$
   - $\forall \sigma_0 \in \Sigma^T,\ \sigma_1' \in \Sigma^{T'},\ \tau(\sigma_0) \rightarrow^{T'} \sigma_1' \implies \exists! \sigma_1 \in \Sigma^T,\ \tau(\sigma_1) = \sigma_1' \wedge \sigma_0 \rightarrow^T \sigma_1.$
3. *Furthermore, P^T is called τ-complete with respect to $P^{T'}$ if it does not contain any superfluous transitions*

- $\forall \sigma \in \Sigma_0^T,\ \tau(\sigma) \in \Sigma_0^{T'}$
- $\forall \sigma_0,\ \sigma_1 \in \Sigma^T,\ \sigma_0 \to^T \sigma_1 \implies \tau(\sigma_0) \to^{T'} \tau(\sigma_1).$

The relations "τ-complete covering" and "τ-complete partition" can be shown to be transitive, anti-symmetric and reflexive and hence describe a partial ordering on the set of possible partitioned systems of P. The ordering is denoted with the $\preccurlyeq_\tau$ operator.

Definition 3.6 (Partial Ordering). *The partial ordering $\preccurlyeq$ amongst partitioned systems P^T and $P^{T'}$ based is defined as*

$$P^T \preccurlyeq P^{T'} \iff \exists \tau,\ P^T \preccurlyeq_\tau P^{T'}. \tag{3.8}$$

Each transition system describes many partitioned transition systems that are complete coverings of itself. The most basic one is the trivial extension which therefore corresponds to the $\perp$ element of the ordering. The $\top$ element is not tractable and more of a theoretical interest. It must cover all possible complete coverings of the trivial extensions for all possible forget functions τ for all possible token sets.

Example 3.1 (Extended System). For this example, consider once more the example program from Listing 2.1. This time, only the control but not the memory state changes, the memory abstraction consists of one single element. Figure 3.1 depicts the basic transition system in ①. The labels of the nodes correspond to the control states and indicate the line number of the program. The graph ② shows the trivial extension P^T with $T = \{t\}$ and the graph on the right represents a partitioning of the trivial extension with $T' = \{t_0, t_1, t_2\}$. The additional two tokens t_1 and t_2 can be thought of as a partitioning of traces depending on which conditional branch has been taken. The forget function $\tau : T' \to T$ maps all $t_i \in T'$ to $t \in T$ (i.e. it *forgets* the index) and therefore the statements $P^T \preccurlyeq_\tau P^{T'}$ and $P^T \preccurlyeq P^{T'}$ hold. ○

For qualitative statements about coverings and reductions one further helper function is necessary.

Definition 3.7 (Semantic Transfer). *The semantic function Γ_τ for two extended transition systems P^T and $P^{T'}$, where P^T is a τ-covering of $P^{T'}$, transfers a function from one system to another by associating the "forgotten" tokens in the covered system with the corresponding "forgotten" traces of the original mapping*

$$\Gamma_\tau : (T \to [\![P^T]\!]) \to (T' \to [\![P^{T'}]\!]) \tag{3.9}$$
$$\phi \mapsto \lambda(t') \cdot \bigcup \{\tau(\phi(l)) | t \in T, \tau(t) = t'\}.$$

Finally, this makes it possible to state the single most important theorem about partitioned transition systems.

Theorem 3.1 (Semantic Adequacy). *If P^T is a τ-complete partitioning or τ-complete covering of $P^{T'}$ the partitioned semantics is adequate (sound and complete), that is*

$$[\![P^{T'}]\!]_P = \Gamma_\tau([\![P^T]\!]_P). \tag{3.10}$$

Simply put, by partitioning a system information may be gained, but it is not possible to construct a less precise set of traces.

3.3 Trace Partitioning Abstract Domain

Now that the groundwork has been laid, it is time to put the pieces together and finally build the trace partitioning abstract domain.

Definition 3.8 (Trace Partitioning Domain). *The trace partitioning abstract domain for a given transition system P contains tuples of the form (T, P^T, Φ) where*

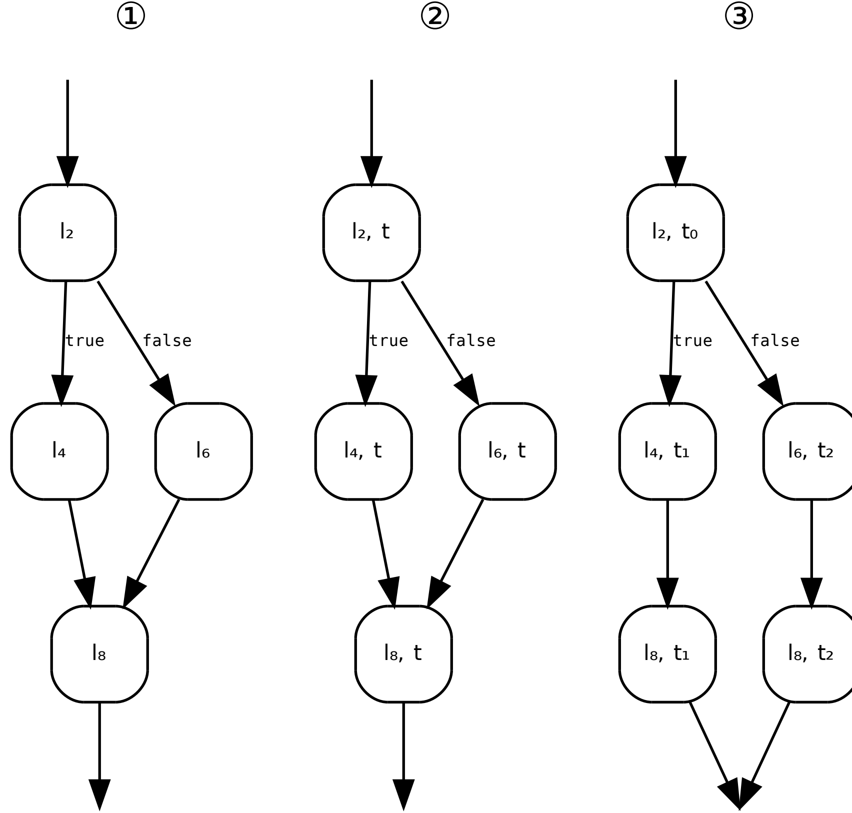

Figure 3.1: Trivial Extension and Complete Partition

- T *is a set of tokens,*
- P^T *is a complete covering of* P *and*
- Φ *is a function* $\Phi : L^T \rightarrow [\![P^T]\!]_P$ *relating partitioned control states with the partitioned traces reaching the control state.*

The ordering can be defined as follows. For two elements $(T, P^T, \Phi) \leqslant (T', P^{T'}, \Phi')$ *if*

- $P^T \preccurlyeq_\tau P^{T'}$ *and*
- $\Phi \subseteq \Gamma_\tau(\Phi')$*, using the semantic transfer from Definition 3.7.*

The choice of Φ is not defined here. It is one more aspect that provides flexibility in trace partitioning. In this context, of special interest are invariants on reachable states of the system. Instead of mapping partitioned control states to traces, the Φ functions that are used subsequently therefore map locations of the transition system to state invariants from some guest domain D, that is $\Phi : L^T \rightarrow D$.

Given an element of the abstract domain, the corresponding state of the concrete domain can be computed using the concretization function.

Definition 3.9 (Concretization Function)*. The concretization of an element* (T, P^T, Φ) *is computed in three steps:*

1. *By projecting* Φ *onto the trivial extension using* Γ_{τ_t}*, where* τ_t *maps all arguments to a single token* t*,*

2. *applying the partitioning concretization γ_δ (Definition 3.2) and finally*

3. *applying the isomorphism τ_ϵ transforming the trivial extension back to the base transition system.*

This can be written more concisely as

$$\gamma_P = \tau_\epsilon \circ \gamma_\delta \circ \Gamma_{\tau_t}. \tag{3.11}$$

To complete the domain, a widening operator has to be defined.

Definition 3.10 (Widening for the Trace Partitioning Domain). *A widening operator ∇_P can be defined by a pairwise widening on the structure and the domain function. Then,*

$$(T_0, P^{T_0}, \Phi_0)\nabla_P(T_1, P^{T_1}, \Phi_1) = (T_2, P^{T_2}, \Phi_2) \tag{3.12}$$

where

- $P^{T_2} = P^{T_0}\nabla P^{T_1}$ *for some widening ensuring* $P^{T_0} \preccurlyeq_{\tau_0} P^{T_2}$ *and* $P^{T_1} \preccurlyeq_{\tau_1} P^{T_2}$ *and*
- $\Phi_2 = (\Phi_0 \circ \tau_0)\nabla_D(\Phi_1 \circ \tau_1)$ *by applying the widening on the composed domain.*

Again, this definition is flexible. What exactly the widening of two partitioned systems is will be addressed when discussing the implementation in Chapter 5. As for the widening on Φ, assuming it has the previously discussed form mapping locations to invariants (i.e. $L^T \to D$), the widening can be defined as the widening of the guest domain.

3.4 Static Analysis

Once more, this report assumes that the domain of interest is formed by the composition of the trace partitioning domain with an invariant domain of the form $\Phi : L^T \to D$ for some guest domain D. This can also be rewritten as $\Phi : (L \times T) \to D$ which is isomorphic to $\Phi : L \to (T \to D)$. This little trick makes it possible to compute the static analysis as a fixed point over the reachable states as previously shown in Section 2.3.

The main difference is that the analysis states are no longer simply the abstract states but consist in fact of a mapping $T \to D$. How to deal with these *partitioned states* is one of the problems addressed when discussing the implementation in Chapter 5.

Chapter 4

Sample

Sample is a tool for static analysis based on abstract interpretation that is currently being developed at *ETH Zürich*. *Sample* is written completely in `Scala`. The project is open with respect to the choice of language as well as the type of analysis. The core of the *Sample* analyzer is split into two major packages which are both part of the `ch.ethz.inf.pm.sample` namespace.

- The `abstractdomain` package contains classes representing the concepts related to abstract interpretation.
- The package `oorepresentation` contains the facilities related to analyzing an object oriented language.

I will start by discussing the former in Section 4.1 before going into details about the specifics of object oriented languages in Section 4.2. This is not meant to be a comprehensive documentation of the *Sample* project but rather a qualitative overview of the architecture providing the foundation for the coming chapters. Unfortunately, there is currently not much more documentation available and, since *Sample* is still under active development, the following material is a snapshot of the current development state. Furthermore, the active development also implies that there are features not yet implemented. The current limitations are briefly discussed in Section 4.3.

4.1 Abstract Domain Representation

The `abstractdomain` package is responsible for handling the abstract domains. A coarse overview of the package is given in Figure 4.1.

4.1.1 Lattice

At the root of the inheritance tree is the `Lattice` trait which, as its name already suggests, represents a lattice as described in Section 2.2. It provides factory methods for elements, the $\top$ as well as the $\bot$ element (`factory`, `top`, `bottom`). Furthermore, the `lub`, `glb` and `widening` compute the least upper bound, the greatest lower bound and the widening of two elements respectively. Most importantly, implementing classes have to provide an implementation for the ordering relation (`lessEqual`). It is discussed first because of its widespread usage throughout *Sample*. There are few classes that do not incorporate or at least interact with this trait.

4.1.2 Values

The class `Expression`, which is not depicted in the figure, represents the result of a statement. This result can have several possible forms amongst others are constants, variables and arithmetic as well as boolean expressions. Although it is not commonly the case, values in *Sample* need not

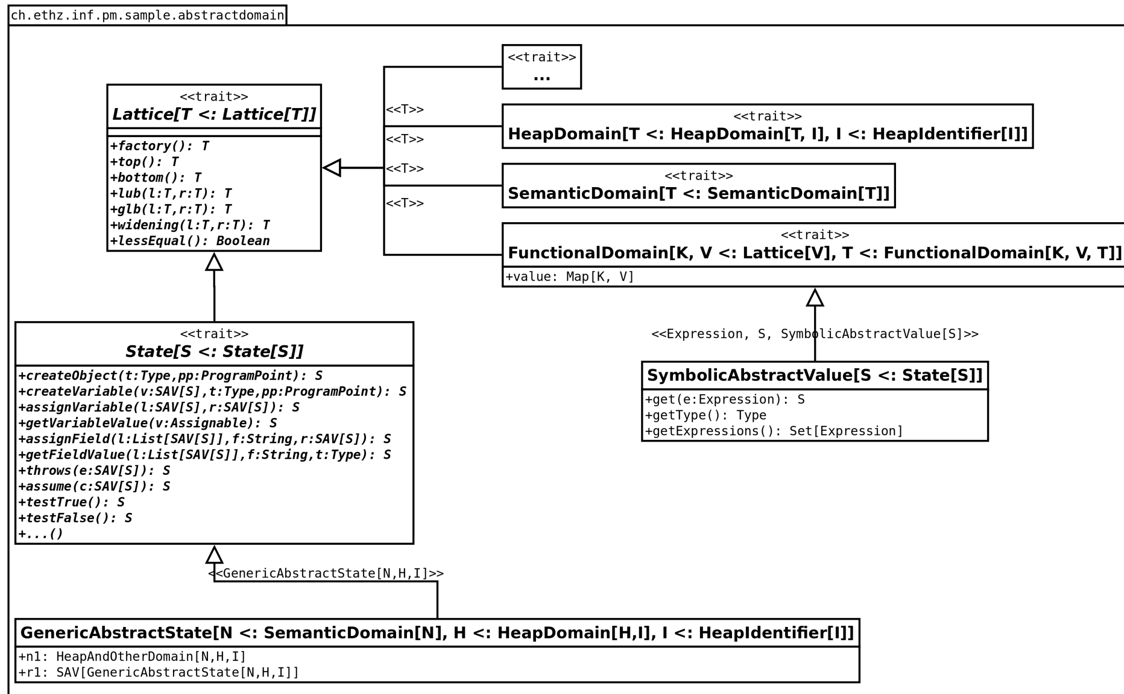

Figure 4.1: The `abstractdomain` Package

be generated deterministically and hence expressions are not sufficient to represent values during the analysis. `SymbolicAbstractValue` (abbreviated `SAV` in Figure 4.1) deals with this subtlety by relating expressions to the states they where generated in. A symbolic abstract value can therefore represent multiple values at once. The following example attempts to clarify this concept.

Example 4.1. Some languages such as `Scala` allow conditionals to return values. Consider the assignment `r = if (x < 0) -1 else 1`. Assuming that evaluating the numerical constant `-1` results in a state `p` and evaluating `1` results in the state `q` respectively, the value representation assigned to `r` will be of the form {`-1` ↦ `p`, `1` ↦ `q`}. ○

4.1.3 States

The `State` trait is the most fundamental trait of the *Sample* project. It represents abstract states of the analysis. A state's behavior is specific to the kind of analysis that is performed and the implementing subclasses have to provide the details.

These details include the basic abstract operations and define what happens when, for example, a variable is created (`createVariable`), when a variable is assigned some value (`assignVariable`) but also what happens when a variable is read (`getVariableValue`) etc.

Furthermore, the state describes what happens when some expression is evaluated in the context of the current state (`assume`). A special case occurs when the current context is evaluated to `true` or `false` respectively (`testTrue`, `testFalse`). This is the mechanism that allows conditionals to be handled by the analysis. Depending on whether the analysis follows the `true`-branch or the `false`-branch during the analysis the branching condition is evaluated accordingly. All these operations have in common that they do not modify the state objects but result in new state objects representing the modified state after applying the corresponding operations.

Since it should be possible to compare and combine states, the trait also incorporates the `Lattice` trait.

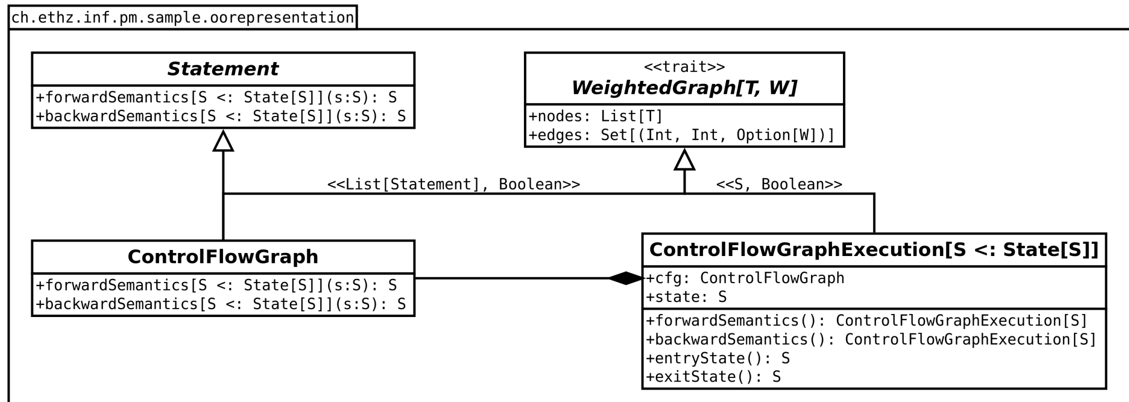

Figure 4.2: The `oorepresentation` Package

The Generic Abstract State

A default implementation of the `State` trait is provided by the `GenericAbstractState` class. This implementation combines two abstract domains. The first one is a domain that keeps track of the heap structure during the analysis. The second can be any other analysis that implements the `SemanticDomain` trait, a simplification of the `State` trait allowing subclasses to safely ignore heap related concepts.

Apart from managing the two domains, the generic abstract state also keeps track of what expression has last been evaluated in the current state in form of the `SymbolicAbstractValue r1`. Keeping track of the current expression not only makes it simple to implement the previously discussed `testTrue`/`testFalse` methods, but also makes it possible to support the assignment of more complex statements (see Example 4.1).

4.2 Object Oriented Representation

The other important package `oorepresentation` is roughly depicted in Figure 4.2. Its main responsibilities are the handling of some object oriented language, this includes the handling of the fixed point iteration discussed in Section 2.3.

4.2.1 Classes, Methods and Statements

Not listed in the figure are the classes representing the standard object oriented concepts. Classes are represented by `ClassDefinition` and provide access like most common introspection frameworks. The class definition provides access to its methods of type (`MethodDeclaration`) which in turn provide access to their execution body as `ControlFlowGraph` object. Instances of these classes are language specific and are generated from implementations of the `Compiler` trait. The *Sample* project currently provides compilers for `Java Bytecode` and for `Scala`.

The second basic notion not entirely depicted in Figure 4.2 is that of a `Statement`. Case classes for assignments (`Assignment`), method calls (`MethodCall`) etc. provide implementations for this abstract class. A list of all subclasses is depicted in Figure 4.3. These implementations of `Statement` mainly provide access to their semantics. The `forwardSemantics` (`backwardSemantics`) takes a state as argument and returns a state representing the analysis after (before) that statement is executed.

4.2.2 Control Flow Graph

At the center of the package are two implementations of the `WeightedGraph` trait representing a generic graph with weighted edges. The trait takes two type parameters specifying the type of the node and

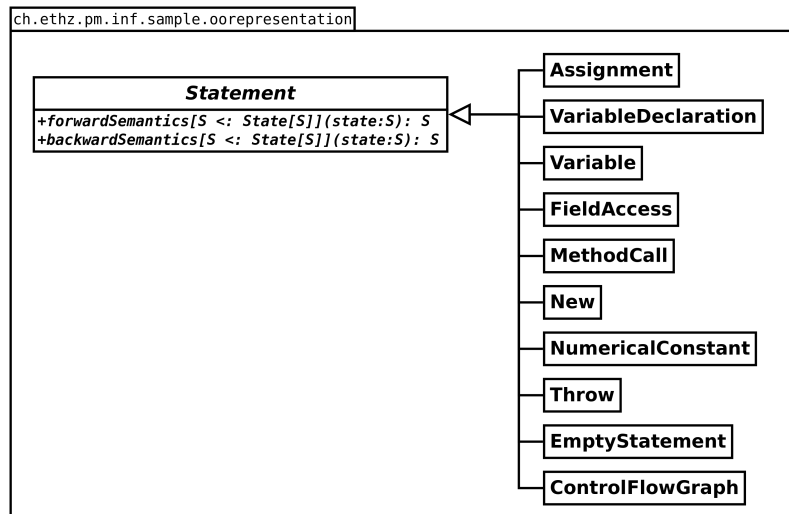

Figure 4.3: The `Statement` Subclasses

the type of the weight of the edges. The first implementation is the `ControlFlowGraph` representing, the control flow graph of the method to be analyzed. The nodes contain lists of `Statement` objects and the edges may be weighted with optional `Boolean` values indicating conditional branches of the control flow. An example of such a control flow graph was presented earlier in Figure 2.1.

4.2.3 Control Flow Graph Execution

The second type of weighted graph is `ControlFlowGraphExecution`. For a given control flow graph and a state, the execution of the control flow can also be represented as graph. Depending on the type of the analysis, the provided state denotes either the entry state (forward analysis) or the exit state (backward analysis) of the system. Then, for each node of the control flow graph containing n statements, the corresponding node in the execution graph contains $n+1$ states. The state $i+1$ of the node then represents the state before statement $i+1$ and after statement i. The edge sets of two corresponding control flow and control flow execution graphs are identical.

The actual computation of the states of the control flow execution is the subject of the next section.

4.2.4 Analysis

The classes presented so far are already sufficient to perform static analyses. A client has to provide a control flow graph, which is usually acquired by compiling some source file and extracting the body of a method of a class. Furthermore, the initial state of the analysis has to be provided. This state is analysis specific but usually some kind of `GenericAbstractState` with a heap and some other domain. *Sample* includes various domains. Amongst those is an interface to `apron` [10], a collection of common numerical domains.

Fixed Point Iteration

The method `forwardSemantics` of the `ControlFlowGraph` computes the exit state of the analysis for a given initial state. It works by returning the exit state of the `ControlFlowGraphExecution` generated by a call to `forwardSemantics` of the same class. This function in turn delegates the work to a private method called `semantics` which is the place where the fixed point iteration happens. The method takes as an argument a function performing a single iteration step and applies it until the states of the control flow graph execution do not change any more or until a widening limit is reached.

The function used in the computation of the forward semantics is called `forwardSingleIteration` and is also a member of the `ControlFlowGraphExecution` class.

This single iteration will be of importance for the discussion of some implementation details of the trace partitioning extension and will therefore be discussed in more detail. A single iteration steps over all nodes of the control flow graph. For each node it computes an entry state and stores it at position `0` in the corresponding execution graph node. The entry state is computed by a helper method (`entryState`) that collects, by means of the least upper bound, all last states of nodes in the execution graph that contain an edge pointing to the current node. If that edge is weighted by a `Boolean` value the corresponding state transformation, `testTrue` or `testFalse`, is applied beforehand.

The missing states of the execution graph node are then computed by the `forwardBlockSemantics`. This method subsequently computes the forward semantics of each statement of the control flow graph block, starting with the previously computed entry state and stores the result in the next element of the execution graph list.

Parameters

The singleton `SystemParameters` specifies various details of the analysis. Most notably, it defines the widening limit (`wideningLimit`) that specifies when the fixed point iteration switches from using the least upper bound to applying the widening, thereby forcing the iteration to converge.

The system parameters also specify the property of interest for the analysis. Properties are of type `Property` and define a method that is automatically passed the final control flow graph execution of the analysis. *Sample* already defines a few properties, such as the `DivisionByZero` property that generates a warning message if there is a possible division by zero.

During the analysis, the parameters singleton keeps track of what class, method etc. is being analyzed at the moment. Furthermore, specifying different output objects (`progressOutput`, `screenOutput`) allows the client to intercept and handle the text output of the static analyzer.

4.3 Limitations

As already hinted at, *Sample* is still under active development. This also means that there are currently several limitations as to what can be analyzed.

There are some limitations as to what language constructs are supported in *Sample*. Interprocedural calls will be supported by the use of contracts. However, the contracts specification language is still under discussion and development. This is especially problematic for languages where native array access is modelled as a method call, which is the case in `Scala`. The problem can be addressed though by "simulating" arrays as will be seen later in this presentation.

Furthermore, not all abstract domain implementations have fully matured yet. Most of the shortcomings are concerned with not fully implementing logic rules such as DeMorgan for negated boolean expressions or detecting contradictions in the internal state. While the former usually just ends the analysis with an exception, the latter can be problematic since it may result in states that are $\perp$ but are not recognized as such.

Chapter 5

Trace Partitioning in Sample

This chapter presents my implementation of the trace partitioning abstract domain and its integration into *Sample*.

Before discussing various implementation details, Section 5.1 illustrates how the concepts of the trace partitioning domain from Chapter 3 are adapted to the concrete implementation. Section 5.2 then continues by presenting an alternative `State` implementation, the core of the extension. The necessary modifications to *Sample* to use this new `State` implementation are the subject of Section 5.3. The flexibility of the implementation is then demonstrated with the various directives in Section 5.4 before Section 5.5 presents the extension of an already existing user interface. Finally, Section 5.6 addresses some of the shortcomings of the current implementation and suggests some future extensions.

5.1 Domain Representation

The main challenge of a trace partitioning implementation lies in representing the elements (T, P^T, Φ) of the trace partitioning domain. This implementation assumes that Φ is of the form $\Phi : (L \times T) \to D$ for some guest domain D. It can therefore be, by curryfication, represented in the form $\Phi : L \to (T \to D)$. This has the advantage that the static analysis can be performed as described in 2.3 using a states of the form $T \to D$.

5.1.1 Tokens

As Mauborgne and Rival suggest, tokens from T represent decisions that have been taken during the analysis along the control flow. Since in the end it is desirable to have easy access to multiple decisions made during the analysis, the singleton tokens used so far are impractical. A more flexible approach is therefore needed.

Definition 5.1 (Token). *A token in T can be represented as a stack of labels from some label set E. A token is then either*

- *the initial token denoted by* `init`*,*
- *a label $e \in E$ or*
- *the combination of two tokens $t, t' \in T$ represented as $t :: t'$.*

The following example illustrates the use of tokens to keep track of decisions and how using a stack representation results in a more intuitive approach.

Example 5.1 (Tokens). Coming back to the extended system $P^{T'}$ from Example 3.1 with the token set $T' = \{t_0,\ t_1,\ t_2\}$, recall the interpretation of the tokens:

- t_0: The initial token, nothing has been decided.
- t_1: The analysis follows along the `true` branch of the conditional.
- t_2: The `false` branch has been chosen during the analysis.

Using the newly introduced notation, the elements can be reinterpreted starting with the initial token `init` instead of t_0. The two tokens discerning the two branches are then `init` :: `If(2,true)` and `init` :: `If(2,false)` respectively. The name of the label (`If`) indicates that a decision has been made about following a conditional. The first argument points to the location of the conditional in question and the second argument indicates which branch has been taken.

Suppose the `true` branch contained another conditional where the distinction of the two branches benefits the analysis. The tokens generated for this distinction will then be the stacks `init` :: `If(2,true)` :: `If(4,true)` and `init` :: `If(2,true)` :: `If(4,false)`. ○

5.1.2 Directives

The tokens need to be generated during the analysis. The mechanism responsible for doing it is called a directive. A directive specifies the kind of distinctions it can make by providing a list of tokens representing possible choices. Applying such a directive to a token t then leads to the tokens that result from pushing each token specified in the directive on top of t. A special case is the directive for merging partitions which, instead of appending tokens to a stack, removes them. The details of this operation will be discussed later.

Example 5.2 (Directive). The `PartitionIf` directive that distinguishes between executions along the two branches of a conditional at position i generates the two tokens `If(i,true)` and `If(i,false)` representing the two possible choices. Applying the directive to some token t results in t :: `If(i,true)` and t :: `If(i,false)`. ○

These directives could be generated from annotations in the source code or, for example, from heuristics. This implementation requires them to be provided externally. The way to do this at the moment is either by means of writing analysis specific code or by using the user interface discussed later on.

5.2 Architecture

The core of the implementation is depicted in Figure 5.1. At the center lies the class `PartitionedState`. The partitioned state represents the full state of the analysis at a given location. That is, at any point $l \in L$ during the analysis the state must keep track of the mapping from tokens to states of the guest domain ($T \to D$).

This mapping can be efficiently represented by a tree structure where the nodes consist of the directives mapping tokens to their children. At the leaves of the tree are the states of the guest domain. This structure is implemented by the abstract class `Partitioning`. Each partitioned state consists of one such partitioning.

As a general rule, the responsibilities of the two classes `PartitionedState` and `Partitioning` are split as follows. The partitioned state's main concern is the implementation of the `State` trait and providing interfaces for directives and for the analysis. The partitioning, on the other hand, solely manages the tree structure and is mainly concerned with the lattice operations.

The directives are represented by the abstract `Directive` class. Implementing classes have to override the `apply` method that is responsible for transforming the tree structure. Furthermore, they have to provide a list of tokens they generate. Since directives are applied between statements, each directive is identified by the program point of the statement it precedes. More details and example implementations will be provided in Section 5.4.

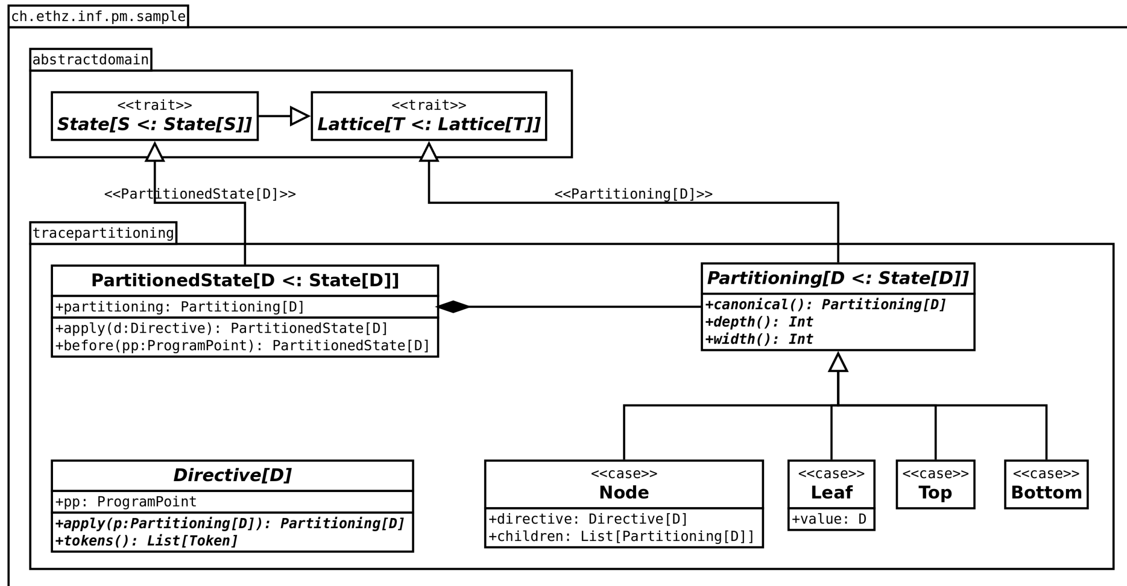

Figure 5.1: The `tracepartitioning` Package

5.2.1 Partitioned State

The `PartitionedState` is similar to the `GenericAbstractState` in that it provides some default implementation of the `State` trait. The type parameter `D` represents the kind of guest domain. This guest domain is typically a generic abstract state.

Semantic Operations

Implementing the `State` trait seems straightforward but comes with its own set of challenges. Apart from the `testTrue` and `testFalse` methods, which require feedback from some directives (cf. Section 5.4), the semantic operations (e.g. `createVariable`) are simply redirected to the leaves of the partitioning.

The main challenge is introduced by the nondeterministic symbolic abstract values. Functions which do not contain abstract values in their arguments are mapped to the leaf states using the private helper function `map` that takes as argument a function transforming a leaf (`f: D => D`) and applies this function to all leaves of the partitioning. Listing 5.1 illustrates the usage of the `map` function.

```
1 override def createObject(t: Type, pp: ProgramPoint): PartitionedState[D] = {
2     map(_.createObject(t, pp))
3 }
```

Listing 5.1: The `createObject` Method

When a method takes a single argument of type `SymbolicAbstractValue`, the situation is already a bit more complicated. An example of such a method is the `assume` method. The expression that the state is supposed to assume could come from several different partitioned states, all containing their own partitioning. The nondeterministic nature of the value makes it necessary to consider every possible combination. This is achieved using the helper function `mapValue` that takes a symbolic abstract value for a `PartitionedState` and a function transforming a leaf with a symbolic abstract value for the guest state type (`f: (D, SymbolicAbstractValue[D]) => D`) as arguments. The way `mapValue` works is that it applies the function for each possible expression and state combination onto the current partitioning and then takes the least upper bound over all results. Two partitioned states are combined

using the zipmap function of the partitioning. This function assumes that partitionings have the same structure and then applies a function combining leaves given as argument to corresponding leaves.

Things get even more complicated when there are two arguments of type SymbolicAbstractValue or even lists of symbolic abstract values. PartitionedState defines private helper functions for all these cases, for more details the reader is referred to the documentation in the source code.

Handling Directives

Another responsibility of the partitioned state is the application of directives. Its apply method delegates the partitioning process to the Directive object only if certain conditions are met. Either

- the directive is a Merge directive (cf. Section 5.4.2) or
- the current partitioning does not exceed a predefined size (determined by width and depth of the tree).

If the conditions are met, a new partitioned state with the transformed partitioning is returned, otherwise the state just returns itself. This mechanism is part of the widening that is globally enforced and not specific the type of directive that is applied. It effectively limits the possible size of the partitioning during the analysis.

5.2.2 Partitioning

The Partitioning represents the tree structure of the partitioned state. Leaves of the tree wrap around a single state of the guest domain. The corresponding subtype is Leaf and its value attribute provides access to the guest state. The Node subclass represents an inner node of the tree. Apart from the reference to the directive that created the node, it also contains a list of its children. The mapping from tokens to children is defined as a one to one relation between corresponding elements of the lists directive.tokens and children. That is, element i of the tokens list of the directive maps to element i of the children list.

As mentioned before, the primary responsibility of the partitioning is defining the lattice operations. Mauborgne and Rival state that the ordering can be defined pairwise on the extended system (using some forget function τ) and the guest domain. They do not present the specifics of their implementation. The description of our implementation follows.

Lattice Elements

The implementation provides two more case classes inheriting from Partitioning, namely the Top and Bottom classes, representing the $\top$ and $\bot$ elements of the lattice respectively. In this implementation, the Bottom element represents the most simple partitioning containing only the $\bot$ state of the guest domain. The tree structure of that element is assumed to adapt to whatever it is compared to. This gives priority to the guest domain over the trace partitioning domain. It furthermore has several practical advantages for the implementation of several directives, a few of which will be further elaborated in Section 5.4.

Ordering

The easiest way to define the partial ordering is the case where a leaf is compared with some other element and then distinguishing the different cases. The code for this can be seen in Listing 5.2.

There are two simple cases where the argument is either Top or another Leaf. The first case simply results in true since any object is less or equal to $\top$, the second case where the tree structure is equal (i.e. both are leaves), the result is that of the comparison in the guest domain. When the argument is Bottom the case is a bit trickier. The special treatment is necessary since it cannot be represented as a leaf. Recalling the definition given earlier the task then becomes trivial, comparing the value element to the $\bot$ element of the guest domain. As for the case when the argument is an inner Node, there

```
1 override def lessEqual(p: Partitioning[D]): Boolean = p match {
2     case Top() => true
3     case Bottom() => value.lessEqual(value.bottom)
4     case Node(_, _) => value.lessEqual(p.lubState)
5     case Leaf(v) => value.lessEqual(v)
6 }
```

Listing 5.2: The `lessEqual` Method in `Leaf`

exists a trivial forget function that maps the argument to a single leaf. This forget function simply forgets all the partitionings. The value of the newly generated leaf is then defined by the Γ function, collecting all leaf states with the least upper bound (`lubState`).

The method comparing nodes is shown in Listing 5.3.

```
1 override def lessEqual(p: Partitioning[D]): Boolean = p match {
2     case Top() => true
3     case Bottom() => children.forall(_.lessEqual(Bottom()))
4     case Node(d, cs) => directive.compatible(d) &&
5         children.indices.forall(i => children(i).lessEqual(cs(i)))
6     case Leaf(v) => false
7 }
```

Listing 5.3: The `lessEqual` Method in `Node`

The trivial cases include again the comparison to `Top` which always returns `true`, and this time the comparison to a leaf, which always results in `false`, since there exists no forget function that can possibly transform a leaf into a node. Consistent with the earlier definition of the `Bottom` element, comparing to the `Bottom` results in checking whether all children of the node are less or equal to the argument. The case where two objects of type `Node` are compared to each other is slightly more complicated. The nodes can only be less or equal to each other if they are compatible. Compatibility is defined by the directive the node contains. In most cases, as this is the default implementation provided in the `Directive` class, compatible simply means equal. Given two compatible nodes, the definition then requires that all children have to be pairwise less than or equal to each other.

Least Upper Bound

The implementation of the least upper bound assumes the commutativity of the operation. The discussion of the greatest lower bound is omitted since it would follow along the same lines.

Listing 5.4 shows the implementation of the least upper bound of the `Leaf` class.

```
1 override def lub(p: Partitioning[D]): Partitioning[D] = p match {
2     case Top() => Top()
3     case Bottom() => this
4     case Node(_, _) => p.lub(p, this)
5     case Leaf(v) => Leaf(value.lub(value, v))
6 }
```

Listing 5.4: The `lub` Method in `Leaf`

Again, four cases are distinguished. Three cases are trivial. Taking the least upper bound with the `Top` element results in `Top`. With the `Bottom` element the result is the current leaf and when the argument is another `Leaf`, the result is simply a new leaf containing the least upper bound of the

two values. Using the assumption of commutativity, the only non-trivial case is delegated to the `Node` implementation shown in Listing 5.5.

```
1  override def lub(p: Partitioning[D]): Partitioning[D] = p match {
2      case Top() => Top()
3      case Bottom() => this
4      case Node(d, cs) => if (directive.compatible(d)) {
5              Node(directive, for ((c1, c2) <- children.zip(cs))
6                      yield lub(c1, c2))
7          } else {
8              case _ => Top()
9          }
10     case Leaf(v) => Node(directive, children.map(lub(_, p)))
11 }
```

Listing 5.5: The `lub` Method in `Node`

This depiction is, for the sake of simplicity, not entirely accurate and a modified version will be presented in Section 5.4.4. First off, the trivial cases for arguments of type `Top` and `Bottom` work as with the `Leaf`.

In case the argument is a `Node`, the distinction is made between compatible nodes and incompatible ones. The former results in a new node whose directive is the current directive and whose children are the least upper bounds of the corresponding children of the current node and the argument. Since it is not obvious what the result of the combination of two incompatible elements would be, the latter case returns `Top`.

Last but not least, arguments that are leaves are passed down the tree structure until they are combined with the leaves of the current tree. This is equivalent to extending the argument into a compatible structure by extending it with the directive of the current node and taking the least upper over the resulting compatible structure as described above.

5.3 Integration

In general, the goal was to keep modifications to the *Sample* code to a minimum in order to integrate the trace partitioning domain. This section describes the core extension as well as how the analysis interacts with the directives in more detail.

5.3.1 Core Modification

The changes made to the core of *Sample* are minimal and limited to a single method in the `ControlFlowGraphExecution` class, namely `forwardBlockSemantics` (cf. Section 4.2.4). A slightly simplified version of the `forwardBlockSemantics` is shown in Listing 5.6.

```
1  private def forwardBlockSemantics(s: S, b: List[Statement]): List[S] = b match {
2      case x :: xs =>
3          val sp = s.before(identifyingPP(x))
4          sp :: forwardBlockSemantics(x.forwardSemantics(sp), xs)
5      }
6      case Nil => s :: Nil
7  }
```

Listing 5.6: The `forwardBlockSemantics` Method

The method takes two arguments, the state of the analysis before the block and a list of statements representing the execution block. It then computes a list of states where each element represents the analysis state after executing the corresponding statement of the block. Two cases are distinguished: When the block is empty, the entry state is returned. When the block is non-empty, the state before the block is transformed by the newly introduced `before` method and prepended to the resulting list. The tail of the block is processed recursively with the entry state obtained from applying the `forwardSemantics` of the current statement on the modified entry state. This one additional state modification is the sole difference to the original analysis code.

The `before` method was added to the `State` trait. It is called to indicate that the analysis is about to process a statement. It gives the state the possibility to react by computing a new state for the remaining analysis. The argument for `before` is a program point identifying the statement (`identifyingPP`) and is obtained by computing the left most program point involved in the statement.

5.3.2 Analysis Interaction

The nature of the fixed point iteration makes tracking the control flow during the analysis a tricky affair. The implementation does not allow for any assumptions to be made about the order in which blocks are analyzed. Furthermore, branching conditions are not evaluated after analyzing a block but while computing its entry state, hindering an intuitive approach to designing directives. Nonetheless, the extension provides facilities that address these issues and, with a bit of practice, make it possible to define effective directives.

Apart from the traditional interaction described in Section 5.2.1 over the `State` trait interface, the `PartitionedState` provides an interface for the additional interaction needed to make decisions based on the flow of control. The first part of this interface is the already mentioned `before` method that informs the partitioned state about what statement is next up in the analysis. Its implementation can be seen in Listing 5.7.

```
1 def before(p: ProgramPoint): PartitionedState[D] = {
2     (this /: TracePartitioning.get[D](p))((s, d) => s.apply(d))
3 }
```

Listing 5.7: The `before` Method

A singleton object called `TracePartitioning` stores all the directives of the current analysis. Its `get` method returns a list of directives for a given program point. The resulting state is then computed by starting with the current state and subsequently applying each directive to the newly obtained state. The fold left operator, represented by `/:` in `Scala`, makes this a one line operation.

In order to keep track of the control flow in the partitioned state, the state also informs directives when a branch has been taken, that is when `testTrue` or `testFalse` have been called. This happens over a simple observer interface (`PartitionedStateObserver`) that all directives inherit. When one of the aforementioned methods is called, all active directives have the opportunity to change the current partitioning. A directive is active if it is present in the current tree structure and its identifying program point coincides with the branching condition that will be evaluated. The implementation assumes that this condition can be accessed by means of the `getExpression` method defined in `State`.

Barring one exception, the two mechanisms give sufficient control over the analysis to implement the directives that are the subject of the next section.

5.4 Directives

This section describes the directives that are currently implemented. Examples demonstrating their usage will be presented in Chapter 6. The illustrations in this section are slightly simplistic looking at states consisting of a single node or leaf whereas in practice the structure of the partitioning might be

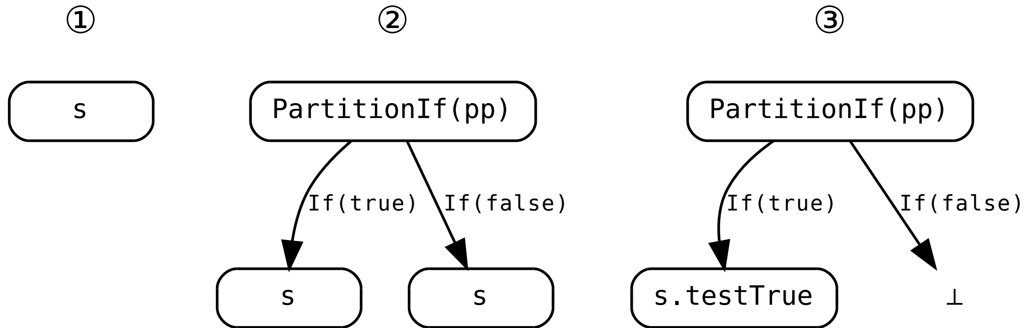

Figure 5.2: The `PartitionIf` Directive

more elaborate. However, the extension to more complicated structures is a straightforward application of the definitions given earlier (cf. Section 5.2.2).

5.4.1 PartitionIf

Since it is the classical example and has already been mentioned, the `PartitionIf` directive will be presented first. Once more, the directive's purpose is to distinguish two kinds of traces based on a conditional flow of control. The first set of traces are those that follow the `true` branch, the second set consists of traces following the `false` branch.

Figure 5.2 illustrates the basic transformations of the directive. Just before the analysis encounters the conditional statement, it will call the `before` method of the partitioned state containing a single leaf with some guest state `s`. The initial state in the figure is labeled with a ①. The state will then look up the directives stored in the `TracePartitioning` singleton and find the `PartitionIf` directive which it subsequently applies to the partitioning. This will result in a tree structure as depicted in ②. The node contains the directive and the left child represents the state following the `true` branch, as indicated by its token `If(true)`, while the right child represents the state following the `false` branch respectively.

During the further analysis, both branches will eventually be taken. When following the `true` branch, the `testTrue` method of the partitioned state will be called and will then be mapped to all the leaves of the partitioning. Subsequently, the state will inform all active directives that the `testTrue` method has been called and gives them a chance to change the partitioning. The `PartitionIf` directive makes use of that facility by setting the child representing the false branch to `Bottom`, representing the contradiction. The resulting state is depicted in ③.

In the subsequent analysis, the `false` branch remains effectively discarded until the two branches are joined together by taking the least upper bound over the over two complementary partitionings. This join operation is depicted in Figure 5.3. The left state `p` is obtained after analyzing the `true` branch, the right state `q` after the `false` branch. Taking the least upper bound results in a new partitioning with the same directive where the least upper bound is applied to corresponding leaves of the tree.

5.4.2 Merge

The `Merge` directive differs from other directives in that it is the only one capable of reducing the size of the tree. It represents the inverse transformation of a given directive, stored in the `source` attribute. Upon application, the directive searches through the partitioning for nodes containing the generating directive and when it finds one, replaces it with the least upper bound of all its children.

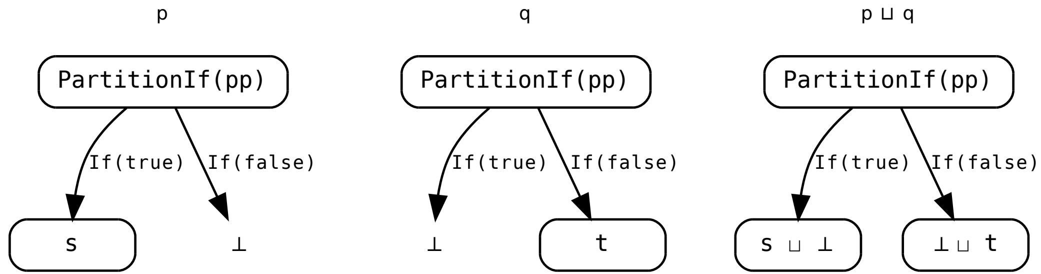

Figure 5.3: Least Upper Bound on Join

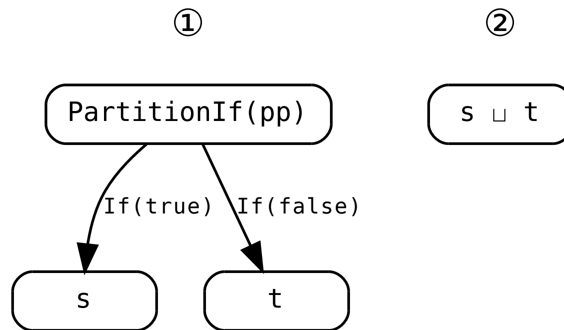

Figure 5.4: The `Merge` Directive

The application of the `Merge` directive for the `PartitionIf` from the previous section is depicted in Figure 5.4.

5.4.3 PartitionValue

The next directive is called `PartitionValue`. Its purpose is to distinguish traces based on the value of a variable.

As Figure 5.5 shows, the implementation is based on the more general notion of a `PartitionCondition` directive. This directive creates a child assuming each condition it stores in form of a list of `Expression` objects.

The `PartitionValue` directive imposes a restriction on what kind of conditions are supported. This restriction is represented by a `VariableContext` object that specifies which variable is restricted (`identifier`) and by what restrictions (`restrictions`). These restrictions are of type `Restriction` which is an abstract class generating an expression. The two implementing subclasses are `Value` and `Range`, representing, for some variable `x` and integers i and j, expressions of the form `x ==` i and j `<= x && x <=` i respectively.

Functionally, there is no difference between the condition and value partitioning except that the latter simplifies the rather involved build up of the expressions.

One more level of specialization is provided by the `PartitionSign` directive that splits an integer value into its three possible sign values.

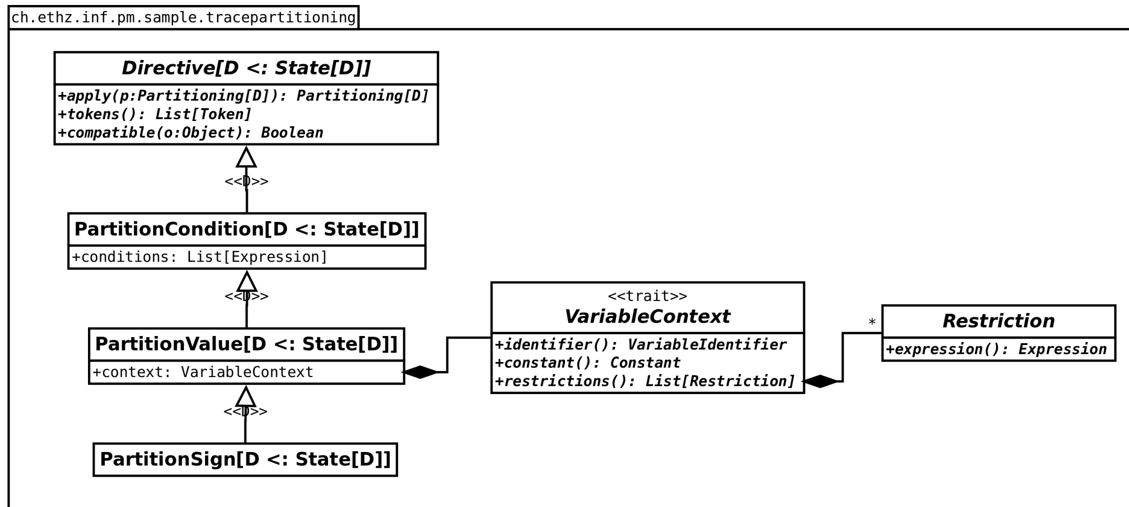

Figure 5.5: `PartitionCondition` and `PartitionValue`

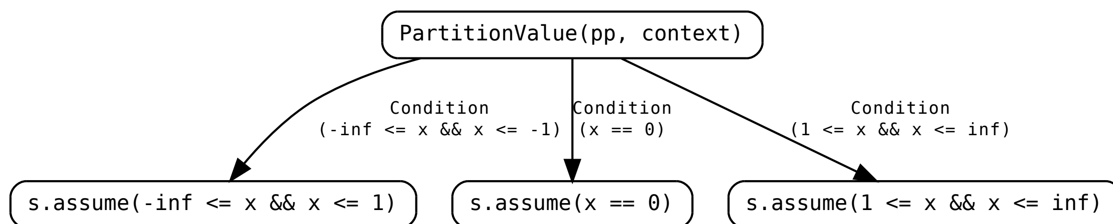

Figure 5.6: The `PartitionValue` Directive

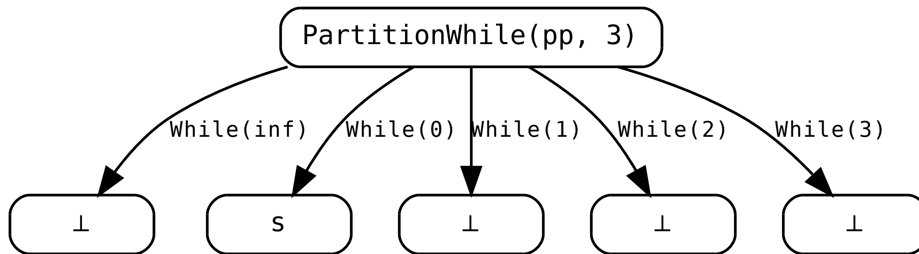

Figure 5.7: The `PartitionWhile` Directive

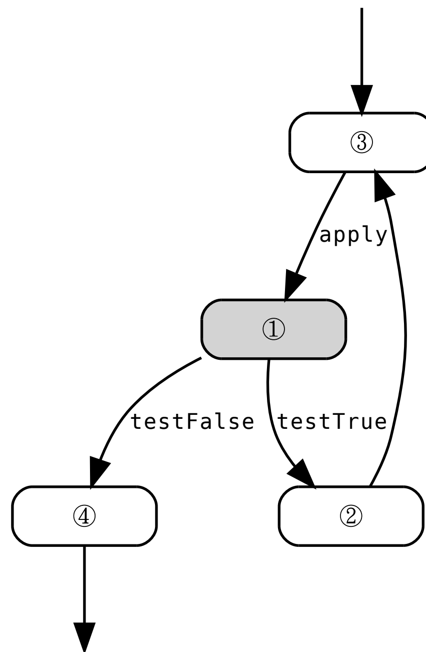

Figure 5.8: Control Flow Graph for a Loop

5.4.4 PartitionWhile

The `PartitionWhile` is by far the most complex directive currently implemented. Unfortunately, its requirements pose some problems to the implementation using the current framework.

Figure 5.7 depicts the initial partitioned state after applying the directive. The directive takes two parameters. First, like all directives the program point, here pointing to the loop condition. The second parameter `n` denotes the number of times the loop is unrolled during the analysis. The resulting partitioned state has `n`+2 children. The first child, identified by the `While(inf)` token, represents the executions that go through the loop more than `n` times. The second child with the token `While(0)` represents the traces that skip the loop completely. The rest of the states with the tokens `While(i)` collect the traces that iterate through the loop exactly `i` times.

Maintaining these invariants during the analysis is quite tricky. Figure 5.8 depicts the major stages involved in analyzing a loop. Here, ① represents the loop condition. Before evaluating the loop condition, the `PartitionWhile` directive is applied. When the condition is evaluated to true, the `testTrue` method is called and the analysis continues inside the loop at ②. The resulting state is then joined with the state coming from outside the loop at ③ and the directive is once more applied to get back to the loop condition. The analysis continues by repeating the previously described steps or by following the `false` branch and leaving the loop structure by applying `testFalse` to ④.

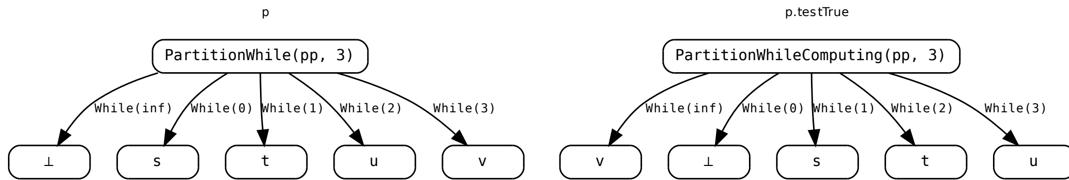

Figure 5.9: The Partitioned State Before and After `testTrue`

Starting with some initial state at ③ and applying the directive initially results in the state depicted in Figure 5.7. Note that applying the directive is required only once. When the partitioning already contains a `PartitionWhile` directive for the loop condition, it does not make sense to generate any further nodes. This is unlike, for example, the `PartitionIf` directive which, inside a loop and lacking a corresponding `Merge` directive, will split leaves with new nodes until the widening limit is reached.

Entering the loop then calls `testTrue` on the state which in turn gives the directive the possibility to modify the partitioning. Having entered the loop means that the invariant for the `While(0)` child is being violated. This state will be analyzed in the loop and hence eventually exit the loop, making it effectively the child that should be mapped to by the `While(1)` token. Shifting the state one child to the right and marking the `While(0)` child with a contradiction restores the invariants for both leaves. The same shift to the right can restore the invariants for the rest of the `While(i)` states in future iterations. The `While(inf)` child needs some special attention. The shift of the `While(n)` will only wrap around if the state for `While(inf)` is `Bottom`. The other case happens when this shift has already happened and the state already represents the traces looping through more than `n` times. The former case is depicted in Figure 5.9 and the code segment handling this piece of logic can be seen in Listing 5.8.

```
1  override def testTrue(p: Partitioning[D]): Partitioning[D] = p match {
2      case Node(d, c) => if (compatible(d)) {
3          val ci = if (c(0) != Bottom()) c(0) else c.last
4          Node(PartitionWhileComputing(pp, n),
5                  ci :: Bottom[D]() :: c.tail.take(n))
6      } else {
7          Node(d, c.map(testTrue(_)))
8      }
9      case _ => p
10 }
```

Listing 5.8: The `testTrue` Method

Although, the handling of this directive is already a bit more complicated than, for example, the `PartitionIf` directive, the real trouble starts with the required non uniformity of the least upper bound operator. Consider the location ③ in Figure 5.8 from the perspective of a partitioned state coming from inside the loop ②. This state will be joined with a state that comes in two possible forms.

1. A state that also contains the `PartitionWhile` directive for the loop condition ①. This is the case if the loop in question itself is inside an other loop and the partitioned state is fed back.

2. Some other state not containing the `PartitionWhile` directive for the condition at ①.

The former case is handled just fine by the default implementation. The least upper bound will result in a new state where the leaves of the directive node are joined pairwise and since all the leaves satisfy the previously stated invariants, the resulting state will satisfy those as well. In the latter case

however, this does not work. The other state represents traces that so far have not traversed the loop. Implicitly, this makes it the leaf of a `PartitionWhile` node for the token `While(0)` and it should be logically treated as such.

It is important to notice that this behavior is specific to the location ③ in Figure 5.8. For states outside the loop, joining two states works as usual. This distinction between the partitioned state inside and outside the loop is addressed by two distinct directives representing either case. The reader may have already noticed in Figure 5.9 that, aside from a shift of the leaves to the right, the directive also changes. The two directives are `PartitionWhile`, representing the general case outside the loop and `PartitionWhileComputing`, denoting the same directive inside the computation of the loop. As already hinted at, the former is transformed into the latter when entering the loop, that is when `testTrue` is called. The reverse happens when the loop is left with the application of the `testFalse` method.

The distinction between the two cases is solely used in the lattice operations least upper bound, greatest lower bound and the widening. For all other intents and purposes they are equal. This fact is reflected by the `compatible` method. Moreover, the `PartitionWhile` directive is the reason why `compatible` was introduced in the first place instead of simply using the default == method.

The `Directive` class has no way of influencing the least upper bound operation of two partitioned states and in my opinion has no business in doing so. However, as just demonstrated, the implementation of the `PartitionWhile` clearly requires to change the way joins operate. Assuming that this directive remains an exception[1], it seems acceptable to have this little piece of logic removed from its natural setting.

Listing 5.9 displays the final implementation of the least upper bound method of the `Node` class.

```
1  override def lub(p: Partitioning[D]): Partitioning[D] = p match {
2      case Top() => Top()
3      case Bottom() => this
4      case Node(d, cs) => if (directive.compatible(d)) {
5          Node(directive, for ((c1, c2) <- children.zip(cs))
6                  yield lub(c1, c2))
7      } else {
8          (directive, d) match {
9              case (PartitionWhileComputing(_, _), _) =>
10                 Node(directive, children.patch(1, List(p.lub(p, children(1))), 1))
11             case (_, PartitionWhileComputing(_, _)) =>
12                 Node(d, cs.patch(1, List(this.lub(this, cs(1))), 1))
13             case _ => Top()
14         }
15     }
16     case Leaf(v) => directive match {
17         case PartitionWhileComputing(_, n) =>
18             Node(directive, children.patch(1, List(p.lub(p, children(1))), 1))
19         case _ => Node(directive, children.map(lub(_, p)))
20     }
21 }
```

Listing 5.9: The final `lub` Method in `Node`

The main differences compared with the simplified version from Listing 5.5 are the additional case distinctions for incompatible nodes. In case two nodes are joined and one of them contains a `PartitionWhileComputing` directive, instead of just returning `Top`, the join is computed as described above. When the argument is a leaf, the method checks whether the current node contains a `PartitionWhileComputing` directive and then performs the join accordingly.

[1] I can not think of any other possible directive with the same requirements

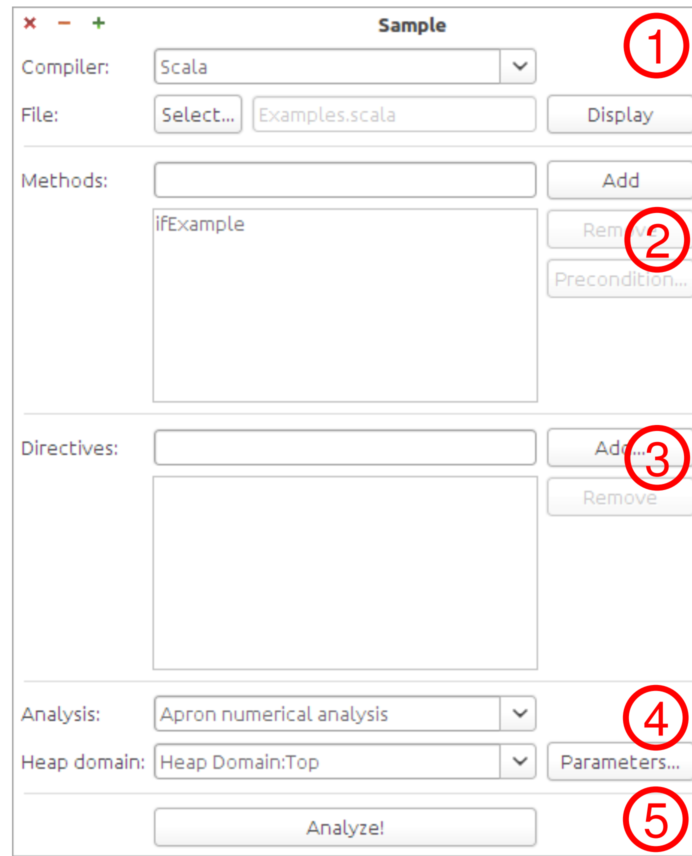

Figure 5.10: The Graphical User Interface

5.5 User Interface

Sample comes with a small graphical user interface (GUI) that greatly simplifies running analyses. This section describes the main functionality of the user interface. I have made various changes both for improving usability in general and to be able to quickly generate directives used in the analysis. The application is written in `Java` and the interface was mostly created using `IntelliJ`'s "UI Designer" plug-in.

5.5.1 Analysis Setup

Figure 5.10 shows a screenshot of the user interface as it is presented after running the application.

The user is asked in ① to first specify the compiler to use and to provide a source file to analyze. The chosen file can be displayed using the "Display" button to the right. The user can then edit a list of methods that should be analyzed in ②. Subsequently, a list of directives follows in ③. The directives can either be entered as a `String`, in which case the `Directive` companion object tries to parse the input, or by means of a wizard discussed later in this section. The next step in ④ is to specify what kind of analysis to run. The choices here depend on the available plug-ins. Furthermore, a heap representation has to be picked. Once everything is set up, the "Analyze" button ⑤ initiates the analysis.

5.5.2 Adding Directives

Pressing the "Add" button in the directives section starts the wizard. An example of a wizard helping to set up a `PartitionValue` directive is shown in 5.11.

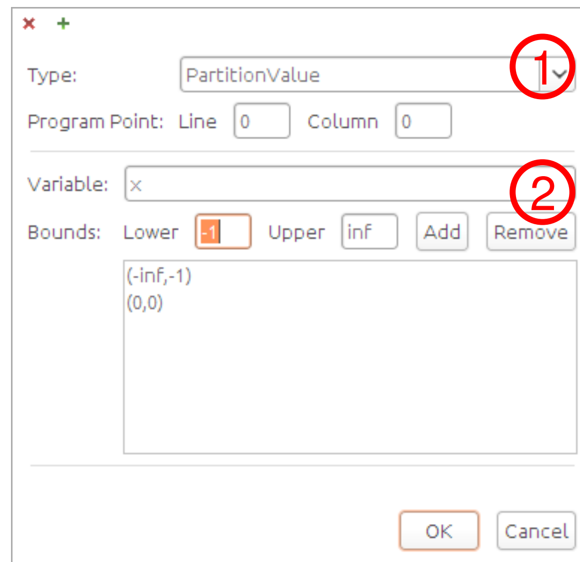

Figure 5.11: Adding a Directive

The top section, indicated by ①, of the wizard is common to all directives and consists of the choice of directive and an identifying program point. Choosing the right program point is currently a tedious task. However, displaying the source inside the user interface using the “Display” button and placing the cursor at the right location provides the line and column numbers in the lower left corner of the window.

The rest of the panel, marked by ②, is directive specific. For the `PartitionIf` directive, no more parameters have to be specified. For `PartitionWhile` only the parameter `n` has to be set. The screenshot shows the additional parameters for `PartitionValue`. The variable over which to partition has to be chosen, here it is `x`. Furthermore, a list of intervals can be provided.

5.5.3 Running the Analysis

Once the analysis is started, the interface might ask some further parameters before actually running the analysis. The `apron` analysis, for example, provides a collection of several abstract domains, one of which has to be chosen. Furthermore, the property of interest for the analysis has to be selected. Again, the available choices are specific to the chosen analysis. For instance, the numerical domains provided by `apron` all support the already mentioned `DivisionByZero` property that generates a warning for every possible division where the divisor can not be guaranteed to be non-zero.

Once these parameters are set, the analysis can be run. Feedback is provided by a progress bar and some status updates. Once the analysis is terminated, a quick log is displayed to the user containing all generated output. This usually includes the warnings of the properties (or absence thereof) as well as some statistics about the analysis, for example, how much time has passed.

A special property supported by most analyses is the `ShowGraph` property. Unlike other properties it does not check any property during the analysis. Upon completion, it displays the control flow graph of the analyzed method. Figure 5.12 shows such a control flow graph. The user can then click on the nodes to get a representation of the control flow graph execution, displaying the sequence of states of the block connected by the statements between them.

5.6 Limitations

The current limitations of *Sample* naturally also apply to the trace partitioning implementation. This includes that calls to methods without contracts result in a total loss of information. Nonetheless,

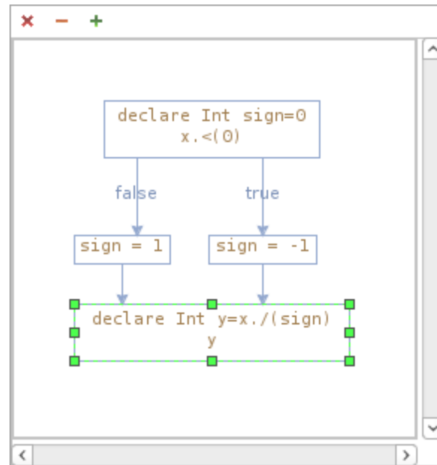

Figure 5.12: The ShowGraph Property

the implementation already includes some facilities to deal with different calling contexts. The Void token and its corresponding directive PartitionNone, for example, can be used to distinguish between different stacks in a state. Further information on the usefulness of this special construct can be found in [11].

Furthermore, the limited availability of numerical types in *Sample* also limits the PartitionValue directive. However, the design of the directive has been made with multiple supported types in mind and the adaption, once more types become available, should not pose a problem.

Generally, the implementation of most directives has been straightforward. I therefore consider the overall design to be quite flexible and hope it will prove easy to extend further. However, there are limitations which became apparent when implementing the PartitionWhile directive. Most of these have already been addressed in the previous section. The directive has an additional flaw which is unavoidable. While in general, the running time and convergence of the analysis with partitioned states depends heavily on how the directives and the widening limits are chosen, the PartitionWhile directive is problematic as soon as non-trivial loops, especially nested ones, are analyzed. This stems from the nature of the invariants imposed on the leaves. Changing the state of the leaf for While(0) affects all other leaves. Changing this one leaf invalidates all other leaves and the iteration computing the loop states has to reach a new fixed point. I speculate that an iteration algorithm aware of the partitioned states might provide some form of mitigation, though this subject is out of the scope of this thesis.

Chapter 6

Evaluation

This chapter will present an adaptation of the examples provided in [11] as well as a short evaluation of the performance impact the trace partitioning has on the static analysis. As previously mentioned, array accesses are not yet precisely supported in *Sample* but required for the following examples. One way to work around this problem is to make pseudo array accesses. Listing 6.1 depicts a normal array access in Scala.

```
1  def arrayAccess(): Int = {
2      val index = 1
3      val array = Array(0, 1, 2)
4      array(index)
5  }
```

Listing 6.1: Normal Array Access

To illustrate what is meant by pseudo array access, Listing 6.2 replicates the same behavior without having to call any access methods and is therefore analyzable using *Sample*.

```
1  def pseudoArrayAccess(): Int = {
2      val index = 1
3      var result = 0
4
5      if (index == 0) result = 0
6      else if (index == 1) result = 1
7      else if (index == 2) result = 2
8      else return 0
9
10     result
11 }
```

Listing 6.2: Pseudo Array Access

The `return 0` statement in line 8 represents the exceptional control flow in case the index is out of bounds. This will result in an exit state and its execution will not interfere with the rest of the analysis. Note that, due to the verbose nature of this access, it is shortened in the following examples wherever logically possible.

Furthermore, since the only numerical type supported by *Sample* at the moment is `Int`, the adapted examples using linear interpolation will not be using floating point numbers as in their original presentation but are limited to integers. However, the principles behind the examples remain the same. Once *Sample* supports `Float`, all that needs to be changed are the type declarations in the guest domain.

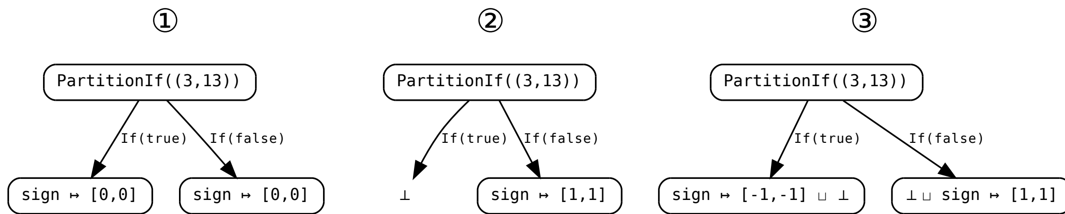

Figure 6.1: Key States of the Analysis

Although the guest domain used in this chapter is once more the interval domain, this need not be the case. It has, however, the advantage that it is a very intuitive domain and can easily be visualized and understood.

6.1 Partitioning a Conditional

The first example here is also the introductory example that was already used to illustrate the extended transition system. The method of interest is shown in Listing 6.3

```
1  def ifExample(x: Int): Int = {
2      var sign = 0
3      if (x < 0) {
4          sign = -1
5      } else {
6          sign = 1
7      }
8      var y = x / sign
9      y
10 }
```

Listing 6.3: The `ifExample` Method

The property of interest is whether or not there could be an unsafe division in this method. That is, is there a division where the divisor could possibly be zero? Intuitively, there is no such division. Proving so, especially using common numerical abstract domains, turns out to be surprisingly complicated.

The division occurs in line 8 and the divisor is the `sign` variable. At the beginning of the analysis, no assumptions about the value of the variable can be made, thus `sign` is represented by $\top$ or, when using an interval domain, the equivalent `[-inf,inf]` interval. After simulating line 2, the value of `sign` is clearly zero, hence it will be represented by `[0,0]`. Continuing, the flow of control is split into the `true` and `false` branch of the conditional. At the end of the branches, the value lies in `[-1,-1]` and `[1,1]` respectively. Joining the two branches just before the division leads to the interval `[-1,-1]` $\sqcup$ `[1,1] = [-1,1]`. This interval obviously contains the zero and the analysis generates a warning when looking at the statement in line 8.

A whole class of commonly used domains, called convex domains, follows along the same line of reasoning and thus fails to prove this seemingly trivial property. More complex domains can provide a solution but are usually prohibitively expensive.

Inserting a `PartitionIf` directive for the condition in line 3 (more precisely at position `(3,13)`) will distinguish traces following either of the conditional branches. Figure 6.1 illustrates the concrete key states of the partitioned analysis.

Up to the conditional statement, the analysis starting with a partitioned state in form of a leaf containing the interval state is completely equivalent to the analysis just described. Once the conditional

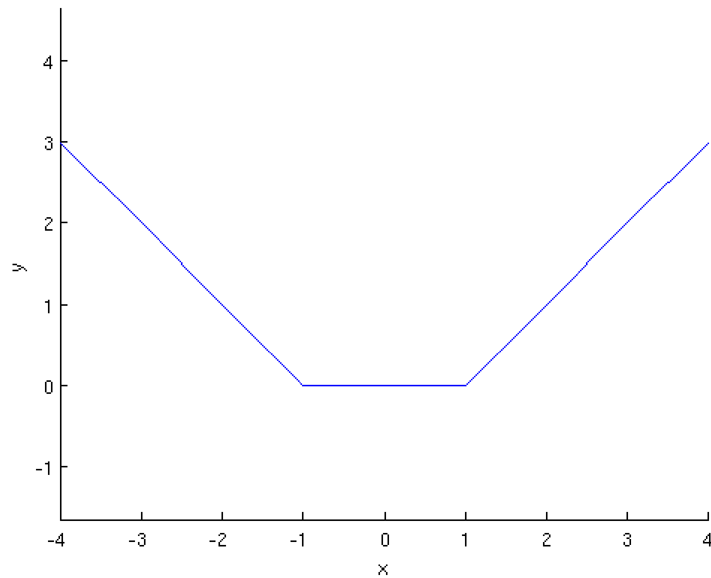

Figure 6.2: The Piece-Wise Linear Function

is encountered, the directive will take effect and split the leaf into a node with two children. The graph ① illustrates the state after analyzing line 3. The partitioned state is then passed through the branches by applying the `testTrue` or `testFalse` method and subsequently the semantic function of the respective branch's single statement. The state at the end of the `false` branch is depicted in ②. The result from the `true` branch looks similar, but with a leaf for the `If(true)` token containing the interval `[-1,-1]`. Before inspecting line 8, the two branches are joined. ③ shows how the guest states are combined leaf-wise using the least upper bound. The analysis then successfully proves that the division in line 8 is safe.

6.2 Partitioning over a Variable

The second example here is concerned with the evaluation of the following piecewise linear interpolation function.

$$f(x) = \begin{cases} -1 - x & \text{if } x < -1 \\ -1 + x & \text{if } x > 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

A plot of the function is given in Figure 6.2. An implementation of this function evaluates the function as $f(x) = c_i + m_i x$, with coefficients determined by the interval x lies in. Starting with the somewhat cumbersome array workaround presented earlier and drastically simplifying it, this method leads to the code displayed in Listing 6.4.

The point of interest in this example is the value of `y` at the end of the method. To have a point of comparison it is once more helpful to quickly step through the analysis using the normal, non-partitioned interval domain. At the beginning, nothing is known. All values are assumed to have the value of $\top$, represented by `[-inf,inf]`. Executing the initial assignments, that is lines 2 to 3, leads to a state where the value of `x` is still undefined and that of `y`, `c` and `m` is `[0,0]`. Since nothing is known about `x`, the four consecutive conditionals are not determined and `c` ends up in the interval `[-1,0]` while `m` is assumed to be somewhere in `[-1,1]`. This information is utterly useless once line 12 is reached because `x` could have any value and the slope could be anything from `-1` to `1`. The analysis therefore

```
def valueExample(x: Int): Int = {
    var y = 0
    var c = 0
    var m = 0

    if (x < -1) c = -1
    if (x > 1) c = -1

    if (x < -1) m = -1
    if (x > 1) m = 1

    y = c + m*x
    y
}
```

Listing 6.4: The `valueExample` Method

concludes that the resulting `y` must lie in the interval given by `[-inf,inf]`. Looking at the plot, this result is disappointing and unnecessarily inaccurate.

Inserting a `PartitionValue` directive at the beginning of the method and a `Merge` directive before returning the result leads to the key states depicted in Figure 6.3. The directive here is applied before the first assignment and distinguishes the three intervals `[-inf,2]`, `[-1,1]` and `[2,inf]` for the variable `x`. Note that since the method works with integers, these intervals cover the whole range of possible values. The initial partitioned state is depicted in ①. The leaves simply represent the assumptions made by the partitioning directive. Continuing with the partitioned state, the analysis gains in precision when accessing the coefficient arrays as is shown in ②, the state right before the polynomial evaluation. The state afterwards, depicted in ③, infers that, for `x` smaller than `-1` or greater than `1`, `y` must be positive, and zero otherwise. Merging the directive results in the leaf depicted in ④. While merging leads to a loss of information, it is still possible to infer that `y` is always greater or equal to zero, which is exactly the purpose of this analysis.

Note that although the intervals for the directive chosen for this illustration coincide with the definition intervals of the linear interpolation function, this is not necessary to gain precision in the analysis. Even a split of `x` into a positive and negative interval would lead to a better result, though it would no longer be possible to prove the lower bound of zero (but that of `-1`).

6.3 Partitioning a Loop

The third example is very similar to the previous one in that it is also concerned with the evaluation of a piecewise linear function. The function in question is defined as

$$f(x) = \begin{cases} x & \text{if } 0 \leqslant x < 2 \\ 4 - x & \text{if } 2 \leqslant x < 4 \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

and plotted in Figure 6.4.

Once more, the linear functions are represented by coefficients c_i and m_i for four intervals stored in an array. The main difference to the example before is how the index for accessing the coefficients is computed. The idea is to store the upper bound of the interval in a special array `iv` and increase the index `i` until `iv(i+1)` is no longer bigger than `x`. This array for the function above would then be `Array(0, 2, 4, 6)`. The regularity of this array was chosen in order to alleviate this presentation of one more pseudo array access than necessary. The final code evaluating the function is shown in Listing 6.5.

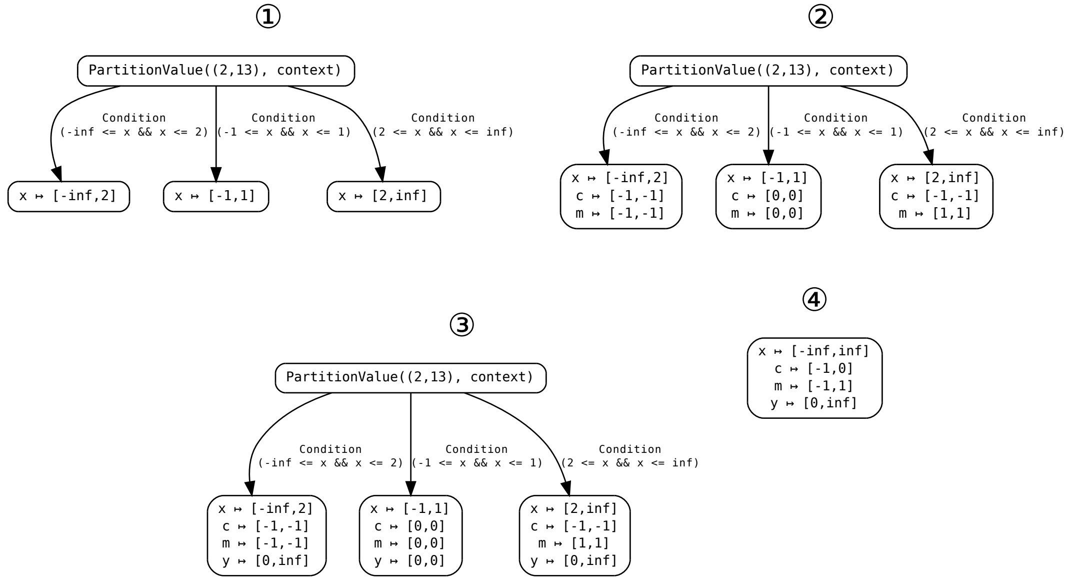

Figure 6.3: Analysis with a `PartitionValue` Directive

Without a partitioned state, the analysis has very limited power. As always, starting with $\top$ for all variables and analyzing the initializing statements `x` has the value `[-inf,inf]` and all other `[0,0]`. The while loop only affects the `i` variable and knowing that the loop is executed at most three times, the analysis will conclude that `i` must be in the interval `[0,3]`. Unfortunately, this leads to all possible combinations for the pseudo array accesses and finally the conclusion that `y` can have any value whatsoever.

Partitioning over the value of `x` does not improve the analysis. Consider the leaf for the token `Condition(2 <= x && x <= 3)`. The state will be analyzed every time the fixed point iteration iterates through the loop. The first time `i` is determined to be in the interval `[1,1]`. The second iteration will then join that result with the newly determined interval `[2,2]` and hence result in `[1,2]`. The last iteration will add 3 to the interval and combined with the state skipping the loop altogether the resulting state will map `i` to `[0,3]`, which is exactly what happened without a partitioning.

The sensible thing to do is therefore to partition over the different executions of the loop statement. Knowing that the array index can have at most four values, distinguishing the traces leading to those four values is a good choice. The state ① in Figure 6.5 shows a simplified version of the state after the loop has been analyzed. The fact that the loop is never executed more than three times is reflected in the state for the token `While(inf)` where it leads to a contradiction. Before evaluating the polynomial, the pseudo array accesses determining the values of `c` and `m` are executed and lead to state ②. So far, the value of `x` has been neglected in this analysis. It is determined by what the underlying domain can conclude from assuming the loop condition. For this example it is `i < (x+2)/2`, which is merely enough to prove that in the end after merging the directive `y` will always be greater than -4. Even with this imprecision caused by the array workaround, the result is a serious improvement over the non-partitioned analysis.

6.4 Performance

It is inherently difficult to make any qualitative statements about the performance of the trace partitioning implementation. The convergence of the fixed point iteration is highly dependent on the inserted directives and the chosen widening limits set for the fixed point iteration as well as for the trace partitioning mechanism.

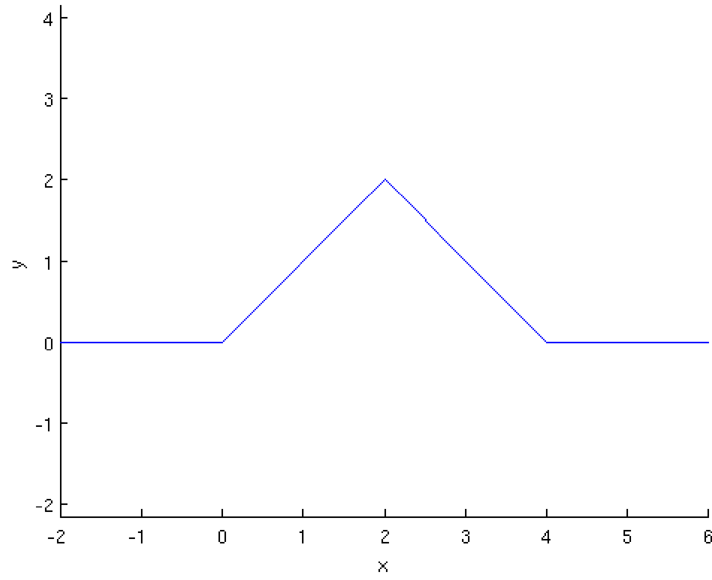

Figure 6.4: The Piece-Wise Linear Function

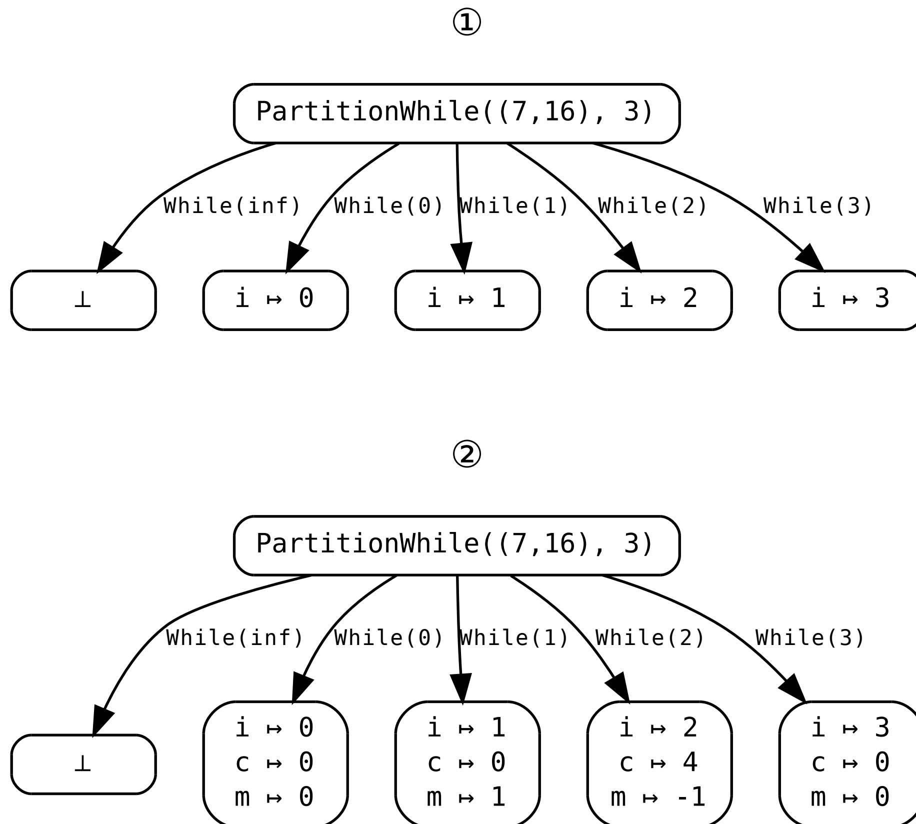

Figure 6.5: Analysis with a `PartitionWhile` Directive

```
def whileExample(x: Int): Int = {
    var y = 0
    var m = 0
    var c = 0
    var i = 0

    while (i < (x+2)/2 && i < 3) {
        i = i + 1
    }

    if (i == 0) { c = 0; m = 0 }
    else if (i == 1) { c = 0; m = 1 }
    else if (i == 2) { c = 4; m = -1 }
    else if (i == 3) { c = 0; m = 0 }
    else return 0

    y = c + m*x
    y
}
```

Listing 6.5: The `whileExample` Method

Ignoring the widening limits, having no directives and performing the analysis with an initial partitioned state in form of a leaf simply adds the constant cost of redirecting the calls to the state interface to the enclosed guest state. On the other hand, having a `PartitionIf` directive inside a loop without a corresponding `Merge` directive leads to an exponential blow-up.

Further complications in estimating the running time stem from the fact that a significant gain in precision can also lead to a significantly faster convergence as was observed by Mauborgne and Rival. Again, this heavily depends on the chosen directives, a problem that is further addressed in Section 7.1.

Using a large body of methods to analyze and some heuristics to automatically generate directives would permit a more useful analysis. Unfortunately, two requirements for this to be feasible are not yet fulfilled. For one, *Sample* is not yet ready to deal with real world code making analysis of a large body of code difficult (cf. Section 5.6). Moreover, the trace partitioning implementation still depends on manually generated directives and is therefore unable to act unsupervised, limiting its applicability to bigger code.

Mauborgne and Rival evaluated their implementation on large projects containing up to 400'000 lines of code using heuristically inserted directives. Their results show a great increase in precision, and a significant decrease of iterations used in some but not all of the analyses.

Chapter 7

Future Work and Conclusion

To conclude, this chapter addresses some of the remaining questions still left open in Section 7.1. Based on the these issues, Section 7.2 proposes possible extensions. Finally, Section 7.3 concludes this report with some final remarks.

7.1 Open Questions

While the implementation provides a solid framework, the bigger picture has so far been neglected. This section tries to address some of the issues that come up when thinking about the future of *Sample* in the context of the trace partitioning extension.

7.1.1 Creating Directives

It is unclear how directives should be generated. The user interface that is currently part of the *Sample* project provides a convenient way of specifying directives for the analysis of small methods but this approach is impractical for larger projects.

The companion object for the `Directive` class contains a parser that recognizes the directives presented in this report. A modified compiler/preprocessor might take advantage of this facility to generate directives from annotations in the code. While annotations are certainly easier to handle than using a user interface, this simply shifts the problem of generating directives to an earlier phase. I personally do not think that the manual generation of annotations has a future. As if having to write annotations was not bad enough, the dynamic nature of trace partitioning does not even guarantee that the directive is actually followed during the analysis. For the programmer unfamiliar with the inner workings of *Sample* this means nothing less than having to deal with additional code clutter whose benefits are highly uncertain at best. Considering how badly annotation mechanisms, even those with clear benefits, are received, it is unlikely that this would ever be adopted in a broader community. If the goal is to make the static analysis accessible to non-specialized personnel the process has to be at least partially automated.

A possible remedy for this problem are heuristics that generate the directives automatically. Rival and Mauborgne propose a few example strategies such as always partitioning the outermost conditional or unrolling the outermost loop to some fixed degree. These two example heuristics are, however, fairly obvious. To come up with a heuristic that automatically inserts a `PartitionValue` is a lot more complicated. For types with limited possible values, that is `Boolean` or `Enum`, it might make sense to just partition into every possible value. For a standard `Integer` this is already prohibitively expensive and makes nesting of directives practically impossible. A slightly more useful strategy would split integers into a negative, positive and zero range. The special handling of the zero might make sense, but other than that it is hard to come up with any convincing reason as to why the analysis should automatically generate such an arbitrary directive.

7.1.2 Integrating Directives

The presented implementation is extensible and coming up with and implementing new directives is not difficult (see Section 7.2). A further challenge will therefore be how to account for this fact, making integration of new directives easy. The design of such a mechanism would, however, be largely dependent on the kind general interface, if any, *Sample* will provide in the future.

7.2 Possible Extensions

Trace partitioning opens up a wide field for possible future extensions. Some of them directly resolve problems presented in the previous section, others build upon the new trace partition implementation.

7.2.1 Heuristics

As described before, heuristics generating directives should be a priority for the future development. The big problem with heuristics is that they are directive specific. Their usefulness might vary among different kinds of software and it is furthermore unclear how different directives influence each other. All these complexities, and of course the complexity of the nature of a heuristic itself, make empirical research inevitable. This in turn will make the development of heuristics both time intensive and probably error prone as well.

7.2.2 New Directives

A further extension would be to provide more directives. The framework is fairly flexible and generating a new directive is as simple as creating a subclass of `Directive`. The more constricted `Partition-Condition` also provides a very flexible mechanism to generate new directives. Recall, this directive splits the current state into leaves where each leaf represents an assumption taken from a list of assumptions. These assumptions are provided in form of `Expression` objects representing arbitrary arithmetic, boolean and, newly introduced at the time of this writing, reference expressions and their negations.

One possible application outside the domain of trace partitioning could be as simple as ensuring the precondition of a method before its analysis with a single expression. Taking this thought further, negating this precondition could also be used to check whether it is actually a necessary requirement.

7.2.3 Domain Specific Directives

The new directives presented so far solely rely on the abstract state interface which is extremely general. Though quite extensive, the `Expression` interface might not be specific enough to represent certain properties of interest. It limits the kind of partitions to numerical, boolean and reference statements. A possible extension could be to look into directives that are specific to the underlying guest domain.

Having a generic abstract state necessarily includes a heap analysis. This could, for example, be used to distinguish traces where some reference is certain to be `null` and traces where it is not. A similar partitioning on the heap domain could discriminate traces where two variables are certainly aliases of each other and traces where that is guaranteed not to be the case.

Considering the diversity of abstract interpretation, the possibilities using this kind of extension seem almost infinite.

7.3 Conclusion

Overall, I am content with the quality of the implementation and documentation this project has produced. There are, however, still a few shortcomings that I would like to address in this section.

7.3.1 Shortcomings

As I have already pointed out, the concrete implementation of the `PartitionWhile` directive does not strike me as conceptually beautiful. Since beauty lies in the eye of the beholder and considering the ridiculous standards most programmers have when it comes to elegance of code, it is a reasonable compromise.

Furthermore, the testing suite for the extension is not quite as extensive as I would like it to be. This strikes me as important since especially the basic operations of the partitioned state are already quite complex and hence an implementation is prone to errors. At the time of this writing the project is still ongoing and more time for testing is definitely allocated. Moreover, the continuous use of the project has improved my overall confidence in the correctness of the implementation. Of course, usage is no substitute for proper testing.

7.3.2 Experience

Looking back, I have collected valuable experiences during this project. The subject has proven to be, especially in the beginning, fairly challenging. The occasional frustration from having to understand complicated, formal mathematical descriptions was usually compensated with a better understanding of what is even after six months still an interesting topic.

The design and implementation gave me a chance to contribute to a bigger project written in a language I was only somewhat familiar with. While demanding, the experience has been rewarding both in terms of learning to work my way through a large code base and at the same time learning a new language.

7.3.3 Contribution

The main contribution of this thesis is arguably more the groundwork for than the actual exploration of new frontiers. However, it is my sincere hope that this report, apart from documenting my project, provides an easy and accessible introduction to the topic of trace partitioning. Furthermore, I hope that the implementation will provide the basis for future research. New subjects to explore are certainly not hard to find.

Bibliography

[1] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *International Conference on Formal Engineering Methods (ICFEM)*, volume 6991 of *Lecture Notes in Computer Science*. Springer, October 2011. To appear.

[2] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics*, pages 138–156. Springer, 2001.

[3] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[5] P. Cousot and R. Cousot. Basic concepts of abstract interpretation. *Building the Information Society*, pages 359–366, 2004.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[7] P. Ferrara. Static type analysis of pattern matching by abstract interpretation. In *Formal Techniques for Distributed Systems (FMOODS/FORTE)*, volume 6117 of *Lecture Notes in Computer Science*, pages 186–200. Springer-Verlag, 2010.

[8] R.W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009.

[11] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[12] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[13] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26–es, 2007.

[14] Wikipedia. Scala (programming language) — wikipedia, the free encyclopedia, 2011. [Online; accessed 16-August-2011].

[15] Wikipedia. Unified modeling language — wikipedia, the free encyclopedia, 2011. [Online; accessed 16-August-2011].