UNIVERSITÀ DEGLI STUDI DI PADOVA

DEPARTMENT OF MATHEMATICS "TULLIO LEVI-CIVITA"

MASTER'S DEGREE COURSE IN COMPUTER SCIENCE

MASTER'S THESIS

Model Checking a Temporal Logic via Program Verification

SUPERVISOR Prof. Paolo Baldan

CO-SUPERVISOR Prof. Roberto Bruni

CANDIDATE Diletta Rigo
ID.2058095

September 22, 2023

Abstract

The thesis explores the possibility of viewing Model Checking as an instance of program verification in order to allow for the reuse of the vast theory and toolset of Abstract Interpretation in the setting of Model Checking. Model Checking is a formal verification technique used to analyse the correctness of software systems, based on a representation of the system as a formal model, such as a finite-state machine or a transition system, and on a representation of the properties it must satisfy as temporal logic formulae. On the other hand, Abstract Interpretation is a program analysis method, based on the idea of extracting properties of programs by (over-)approximating their semantics over a so-called abstract domain, typically a complete lattice, whose elements represent program properties. The thesis focuses on ACTL, the universal fragment of the temporal logic CTL, which can describe properties of executions which are universally quantified. It shows how properties expressed in ACTL can be mapped into programs written in a suitable programming language, whose semantics consists of counterexamples to the validity of the formula. Then such a program is analysed by Abstract Interpretation over some abstract domain, exploiting the idea of local completeness as put forward in some recent work, combining lower- and under-approximations.

"If you want to make God laugh, tell Him about your plans."

To second chances.

Acknowledgements

The first thank and my deepest gratitude go to Professor Paolo Baldan and Professor Roberto Bruni, for guiding me in this thesis work with trust and patience, and for their constant and invaluable feedback.

This journey would not have been possible (and certainly would not have had the same meaning) without the support of the many people I have been fortunate to have by my side along the way. Hoping to have conveyed to them in everyday life at least some of the sincere gratitude I feel, I still want to extend my heartfelt thanks to them.

To mom and dad, for always being there.
To my grandparents, for the light they shed on my path.
To Maria Camilla, Giacomo and Emanuele, for being the best housemates and siblings.
To Alessandra, Corrado and Giovanni, for having me as part of their family.
To Nicolò and Filippo, for the synergy of group work and all the help in tinkering.
To Anna and Silvia, for their sweetness and grace.
To Corollario and Gruppo Giovanissimi, for giving meaning beyond purpose.
To Damiano, the wise man on the mountain I can turn to for advice.
To Giacomo, for the richness of the shared discussions and experiences.
To Nunzio, the older brother I am so grateful I have found.
To Pietro, for proving me every day that love is unbounded.

Contents

Introduction

Software plays a key role in every aspect of our lives, with a continuously increasing massive presence and heterogeneity; moreover, we witness a growing awareness of its social and environmental impact, which is demanding requirements for safety and reliability certifications. The growing complexity and size of software make it necessary to invest more in these aspects: errors can be very expensive in monetary terms, but also in terms of human lives, so they must be prevented as much as possible in different areas of application. Early detection is also crucial, as the cost of software failures after mass deployment is tremendously higher than at preliminary stages.

Notable examples of hardware-related bugs are Intel's Pentium-II bug in the floating-point division unit or malicious codes Meltdown [23] and Spectre [22], which exploit critical vulnerabilities in modern processors that allow programs to steal data. One of the most costly software bug is related to space travel: the crash of the Ariane-5 rocket due to a number format conversion error, which could have been avoided at design phase, helped in bringing to public attention the risks associated with the usage of complex computing systems, resulting in increased support for research devoted to ensure reliability of safety-critical systems. In particular the automated analysis of the Ariane code [12] was the first example of large-scale static analysis based on Abstract Interpretation [11].

Formal Verification. The software certification task, which is crucial to ensure the sustainability of the digital ecosystem, cannot be accomplished with testing alone, but must be performed with rigorous tools for formal verification, which can provide mathematical proofs of soundness. To quote Dijkstra:

> *Program testing can be used very effectively to show the presence of bugs but never to show their absence.*

Formal verification is however extraordinarily difficult to perform: we must face intrinsic limitations due to fundamental negative results (undecidability due to Rice's theorem) together with complexity and scalability issues, which prevent applicability to industrial-sized code even when the task is theoretically feasible; real-world applications require on one hand the ability to handle increasing levels of complexity, on the other to deal with quantitative and probabilistic aspects and uncertainty.

Abstraction. One method to address the complexity of the problem is through the use of *abstraction*. Abstraction involves eliminating or simplifying irrelevant details that are not crucial to solving the problem or verifying the desired property. Verifying the *abstract* model is more efficient, however it could yield spurious outcomes, i.e. it could indicate the presence of a non-existent error (false negative) or miss the presence of an actual error (false positive). Therefore, in abstraction we face a tension between efficiency and precision: a coarser model is simpler to analyse, however it may be uninformative; on the other hand a richer abstraction could be more precise but too expensive at the computational level. Luckily, error can be kept one sided: if we *over-approximate* the system behaviour, we admit *false negatives*, i.e. erroneous counterexamples to the validity of a formal specification. Conversely, if we *under-approximate* the system by removing irrelevant behaviour, we introduce *false positives* and we cannot conclude that the specification is met, but ensure that any error detected in the abstract system is a true error in the original system. In program verification we typically deal with over-approximations, which are useful to prove safety properties of programs (avoidance of error states) but could raise false alarms; on the other side, under-approximations are useful to detect true errors (but could fail to detect all the errors in the software) and to verify liveness properties, i.e. properties that state that some desired good behaviour is accomplished. These approaches relate to two theoretical properties of the abstraction: *soundness* (correctness of the abstraction implies correctness of the original system) and *completeness* (if the abstraction is incorrect, there exists an error in the original system).

Building on the above conditions, the work in the thesis stands at the convergence of two classical formal verification techniques, Model Checking of temporal logic specification and Abstract Interpretation, and aims to provide new insights from their symbiosis.

Model Checking. Model Checking (MC) is a formal verification technique first introduced in [4] used to analyse the correctness of systems, based on a representation of the system as a formal model, such as a finite-state machine or a transition system, and on a representation of the properties it must satisfy as temporal logic formulae. The main technical challenge is the state-explosion problem: as the number of state variables in the system increases, the size of the system state space grows exponentially. In order to tame state explosion, several symbolic or approximated approaches have been considered, such as Symbolic MC, Abstract MC, and Bounded MC [5, 7, 17]. In particular the driving idea of symbolic MC is to use symbolic representations to manipulate sets of states and transitions, and then use logical operations to reason about these sets efficiently.

Model Checking is an example of a technique in which abstraction could be used to simplify the system: in [18] a seminal Counterexample-Guided Abstraction technique (CEGAR) is introduced in order to automatise the abstraction generation by iteratively extracting information from *false negatives* due to over-approximation. As said, over-approximation is *sound* for safety properties, i.e. no false positives are found (if a property is verified in the abstract system, then it holds also for the original system). This method is also *complete*, meaning that no false negatives are found, for an important fragment of ACTL*, a widely used temporal logic. More precisely, CEGAR is complete for the fragment of ACTL* that admits counterexamples in the form of finite or infinite traces (i.e. finite traces followed by loops) [18].

Temporal logics. A temporal logic is a formalism to reason about proposition qualified in terms of time. In the discrete time case, we focus our attention on the order in which properties hold: the formalism let us express whether a property holds in the current state, in the next state, in every state from now on, or if there is some state reachable in the future in which the property will hold. The logics we will be dealing with are propositional logics to which we add temporal modalities and path quantifications [26]. Temporal modalities will be used for the expression of both linear-time properties and branching time properties; linear-time properties consider the system moving along a single path, while quantifications over paths must be added to describe branching-time properties, which consider the system moving on all possible paths.

Abstract Interpretation. Another important formal verification technique in static analysis is Abstract Interpretation, which again uses abstract models of program's behaviour to verify and analyse properties of the program itself. Abstract Interpretation was first introduced in [2, 3] as a sound-by-construction method for verification: the problem which has to be addressed for the precision of the analysis is again the *completeness* of the Abstract Interpretation, which is closely related to the goal of deriving the most abstract domain to decide program correctness without raising false alarms. Completeness intuitively encodes the greatest achievable precision for a program analysis on a given abstract domain, but unfortunately the most abstract refinement, which comes as solution of a recursive domain equation [14], as currently known yields an abstract domain that is often way too fine grained: as observed in [21], the completeness refinement may result in a too concrete abstract domain, hence making the abstract analysis useless, to the point of coinciding with the concrete domain.

In particular, in the setting of Abstract Interpretation, global completeness for (sound by design) analysis of generic software (and Turing complete languages) has been shown to be hard to realise [21]: some recent work [32] introduces the weaker concept of *local completeness* (local in the sense that analysis is performed with respect to specific initial states) and builds on this idea to eliminate false alarms in sound analysis: a program logic LCL_A, parametric in the abstract domain A, is then introduced to this goal. The proof system is designed to combine under- and over- approximations in order to simultaneously check both program correctness and incorrectness. Some effort has been spent in giving bounds to the number of false-alarms raised, introducing the notion of *partial completeness* as a weakening of precision in [30].

Local Completeness and Repair. In [29], the authors introduce a novel approach to domain refinement by leveraging the concept of local completeness proposed in [32]. Refinement, i.e. the process of adding (or removing) elements to the abstract domain to increase precision (preserving soundness), can be performed on a rough abstraction in order to add only the necessary information, alternating analysis and refinement in a forward or backward inspection of the program. This approach introduces algorithmic techniques that aim to optimally repair any local incomplete abstract analysis by adding elements to the the domain whenever a local completeness proof obligation is violated. In this framework, the abstract domain is replaced by the pointed-shell, i.e. the most abstract refinement that is locally complete. This Abstract Interpretation Repair (AIR) approach shares similarity with the already cited CEGAR, which is shown to be an instance of AIR ([29, Sec. 6]). This intertwining approach is also suggested in [25], which shows that a model checker can be formally designed by calculus, by Abstract Interpretation of a formal trace semantics of the programming language.

Model Checking as Abstract Interpretation

The idea of reusing ideas and techniques from the setting of Abstract Interpretation for making Model Checking more effective has been explored by several authors. In particular some apply abstractions technique to the μ-calculus: [9] extends Abstract Interpretation to the analysis of both existential and universal reactive properties, showing how abstract models may be constructed by symbolic execution of programs, [13] deals with a generalization of the μ-calculus and identifies a relatively complete sublogic, [19] presents a novel game-based approach to abstraction-refinement for the full μ-calculus, while [27] develops a theory of approximation for systems of fixpoint equations in the style of Abstract Interpretation, showing that up-to techniques can be interpreted as abstractions. Other works relate more in general to properties transformations: [8] uses simulations parameterised by Galois connections that relate the lattices of properties of two systems to study property-preserving transformations for reactive systems, [16] studies the standard means for relating a specification to its refinement and for relating an implementation to its abstraction using Kripke structures and characterizing the classic tools of Galois connections in terms of binary simulation relations that possess desirable structural properties, and [6] introduces a novel logic for the introduction of nondeterministic and concurrent processes expressed in a process algebra, which allows for compositionality to verify correctness.

This thesis takes a different perspective, which consists in viewing Model Checking directly as an instance of program verification in order to allow for the reuse of the vast theory and toolset of Abstract Interpretation in the setting of Model Checking. Given a specification logic, the idea is to map properties expressed in the logic into programs written in a suitable programming language, whose semantics consists of counterexamples to the validity of the formula. In particular, we combine under- and over-approximation based on local completeness.

ACTL. The thesis focuses on ACTL, the universal fragment of the temporal logic CTL, which can describe properties of paths which are universally quantified. Given a set of basics propositions, properties φ on states that can be expressed - apart from basic propositions, conjunction and disjunction of properties - are "for all possible paths, next φ" ($\mathsf{AX}\ \varphi$), "for all possible paths, eventually φ" ($\mathsf{AF}\ \varphi$), "for all possible paths, always φ" ($\mathsf{AG}\ \varphi$), "for all possible paths, φ_1 until φ_2" ($\varphi_1\ \mathsf{AU}\ \varphi_2$). Some invariants expressible in this logic are for example the fact that a system does not reach a deadlock, or that it never happens that two traffic lights at an intersection are green at the same time.

The first step towards our contribution is the identification of a language, called mocha (for Model Checking as Abstract Interpretation) over which ACTL formulae can be mapped in a way that for a given formula φ and system $(\Sigma, \rightarrowtriangle)$, the semantics of the program associated to φ over the system consists of the counterexamples to the validity of the formula (states $\sigma \in \Sigma$ which do not satisfy the formula φ). The language, based on Kozen's Kleene algebra with test [10], consists of a set of basic operators which allow to extend, filter, disregard computation paths, which can be combined with the usual regular commands of sequential composition, join (choice command), and Kleene iteration.

Concrete Domain and Semantics. In this framework, the concrete domain must retain enough information to allow computing counterexamples while executing programs: a concrete state for mocha is built as a *stack* of the form $\langle \sigma, T\rangle :: S$, where $\sigma \in \Sigma$ represents the current state of the original system, $T \in 2^{\Sigma}$ represents the set of traversed states, $S \in (\Sigma \times 2^{\Sigma})^*$ represents the rest of the stack. We chose to use finite stacks of this shape for the recursive verification of nested temporal formulae and for preserving traces, which ACTL formulae refer to. One can imagine that $\langle \sigma, T\rangle :: S$ represents all traces starting at σ.

The concrete semantics is then defined for the basic expressions $\mathsf{p}?, \mathsf{loop}?$, next, post, push, pop, that let us express ACTL formulae, as follows: $\mathsf{p}?$ checks the validity of the proposition p on the current state σ, $\mathsf{loop}?$ checks if the current state σ loops back to one of the states in the current trace T, the expressions next and post extend the trace by one step in all possible ways, but next does not keep track of this in the trace T, while post does. The expressions push and pop act on the stack respectively by extending the current stack to start a new analysis on $\langle \sigma, \varnothing\rangle$ and by restoring the previous trace from the stack. The concrete elements for the computation are *sets* of stacks, so the collecting semantics is given as the standard additive lifting of the semantics defined on a single stack.

Programs as regular commands. To each formula φ of ACTL we then assign a program $\lfloor\overline{\varphi}\rfloor$ that computes counterexamples to φ, i.e. the semantics of the program $\lfloor\overline{\varphi}\rfloor$ starting from a set of states will filter those states in which φ does not hold. For example, the basic case for a basic proposition is $\lfloor\overline{\mathsf{p}}\rfloor = \neg\mathsf{p}?$, while for instance a counterexample to $\mathsf{AF}\ \varphi$ is defined as

$$\lfloor\overline{\mathsf{AF}\ \varphi}\rfloor = \lfloor\overline{\varphi}\rfloor; \mathsf{push}; (\mathsf{post}; \lfloor\overline{\varphi}\rfloor)^*; \mathsf{loop}?; \mathsf{pop}$$

which means that we find a counterexample to $\mathsf{AF}\ \ \varphi$ when φ does not hold in the current state ($\lfloor\overline{\varphi}\rfloor$) and after that there is a maximal trace ($\mathsf{loop}?$) that traverses reachable states that do not satisfy φ (they are collected using the command $(\mathsf{post}; \lfloor\overline{\varphi}\rfloor)^*$).

Injecting Abstraction. We propose three different abstractions that allow for abstract Model Checking. A first abstraction on stacks is complete, and it consists in a transformation of the concrete domain that can be used together with coarser abstractions: this can be done thanks to a more general result, which states that the composition of a globally complete and a locally complete abstraction is again locally complete (with respect to the same element and the same operation) if some commutation relations hold. This abstraction aims at simplifying the representation of the concrete elements by merging stacks that share the same current state in the head and the same tail of the stack: informally we could say it merges pasts relative to compatible histories.

Then we propose two ways of lifting an abstraction on states $(\alpha_\Sigma, \Sigma, A_\Sigma, \gamma_\Sigma)$ to an abstraction on stacks $(\alpha, \mathsf{C}, \mathsf{A}, \gamma)$. The first of these two approaches is based on partitions: abstract elements are set of stacks whose states are equivalence classes of the states in the original concrete system. We show how the abstraction is induced from the partition, and how partition-based abstractions are compatible with the previously introduced past-merging abstraction.

The second approach aims at having as abstract elements single abstract stacks, instead of *set of stacks*: it is built by merging all the states, all the traces and all the tails of the stacks in a concrete set.

For both approaches we analyse which are the conditions to have completeness for basic expressions of the language mocha.

A Local Completeness Logic The above approaches are sound, but they can provide false alarms if they are not complete. The idea is to use the notion of local completeness developed in [32], focusing on a single execution trace produced by the application of abstract transfer functions on some input of interest. In particular, we study conditions on Best Correct Approximations of basic expressions (i.e. the smallest correct abstractions of the basic expressions of the language) and present examples of abstractions by looking at *local* conditions, exploiting the proof obligations required in the Local Completeness Logic. In fact, the proof obligations needed to derive the statements of the logic are local obligations because they are relative to a specific computation trace. Since logical derivations cannot work with locally incomplete abstractions when some proof obligation fails, the abstract domains need to be repaired to achieve local completeness, and the application of the ideas introduced in [29], to use these failing conditions to guide the identification of effective abstraction refinement techniques, is the natural next step for which this thesis lays the foundation.

Thesis structure

The first part of the thesis is devoted to recalling the necessary background: Chapter 1 is dedicated to Model Checking, and we recall some basic notions regarding transition systems and modal temporal logics, with a specific focus on ACTL and the CEGAR tool. Chapter 2 is devoted to Abstract Interpretation: we recall notions of order theory, Galois connections and closures, then we introduce the Kleene regular language on which the Local Completeness Logic and the mocha are modeled. In Section 2.2 we focus on the more recent notion of Local Completeness and introduce the Local Completeness Logic.

The second part of the thesis is the original contribution: in Chapter 3 we introduce the mocha language and give the translation of ACTL into it, illustrating some computations examples and proving the relation between ACTL formulae and mocha programs. Chapter 4 is dedicated to the study of abstractions that can be injected onto the concrete domain: in Section 4.1 we study a complete abstraction and prove a general result about the preservation of local completeness by composition of abstractions; in Sections 4.2 and 4.3 we present two different ways of lifting an abstraction on states to the domain of stacks. Finally, in Section 4.4 we present a brief account of AIR for its application to our setting.

Part I

Background

Chapter 1

Model Checking

1.1 General Concepts

Model checking is an automated formal verification technique that, given a (typically finite-state) model of a system and a formal property to be verified, systematically checks if the property holds for (a given state in) that model. Let us try to break this definition down: first of all, Model Checking is a verification technique that is based on *models*, that can be defined as descriptions of the system behaviour in a mathematically rigorous and unambiguous manner. The step of giving a model of a system is already useful in detecting possible errors and inconsistencies.

The aim is then to verify a *formal property*, i.e. the specification of the correct behaviour given in a formal language which allows one to express in a precise and unambiguous way the properties to be verified. The *systematic check* is done in the first instance as a brute force exploration of all possible systems state, thus posing the challenge of exploring bigger and bigger state spaces: with the help of clever algorithms, data structures and representation, the check can be refined to handle larger state spaces.

While the model description addresses *how* the system behaves, the properties prescribe *what* the system should or should not do. As said, the model checker examines all relevant system states to check whether they satisfy the desired property: if a state that violates the property is singled out, the model checker often provides a *counterexample*, i.e. an execution path from the initial state to the violating state, that indicates how the model could reach that undesired state. This information can then be used for debug and correction of the system and the model.

Models of systems describe their behaviour, and usually are built as finite-state graphs, called transition systems or Kripke structures, in which the nodes represent the *states*, which comprise some kind of information (values of variables, previously executed statements, properties that are true in that state, ...), and edges represent *transitions*, which describe how the system evolves from one state to another.

Specifications, that should be precise and unambiguous, are usually stated in a property specification language; we will focus on different kind of *temporal logic*, a form of modal logic that is appropriate to specify relevant properties of ICT systems, adding to the traditional propositional logic operators that refer to the behaviour of systems over time (in particular to the *order* in which properties are true or false). In the terms of mathematical logic, Model Checking verifies that the system description is a *model* of a temporal logic formula.

1.1.1 Transition Systems

A *transition system* is an unlabeled, finite, directed graph, where nodes model states, that may or may not satisfy certain properties in a given set, and the edges represent transitions denoting the state changes. More precisely, we have:

Definition 1.1 (Transition System)**.** A **transition system** is a tuple $(\Sigma, \mathrm{I}, \mathbf{P}, \rightarrowtriangle, \vdash)$ where

- Σ is a finite set of states ranged over by σ;
- $\mathrm{I} \subseteq \Sigma$ is the set of initial states;
- $\rightarrowtriangle \subseteq \Sigma \times \Sigma$ is the *transition relation*;
- $\mathbf{P}$ is a (finite) set of basic propositions, ranged over by p, which includes the trivial proposition tt such that $\sigma \vdash \mathsf{tt}$ for any $\sigma \in \Sigma$ and we write ff for $\neg\mathsf{tt}$.
- $\vdash \subseteq \Sigma \times \mathbf{P}$ is the satisfaction relation, where for $\sigma \in \Sigma$ and $\mathsf{p} \in \mathbf{P}$ we have either $\sigma \vdash \mathsf{p}$ or $\sigma \nvdash \mathsf{p} \cong \sigma \vdash \neg\mathsf{p}$.

The behaviour of the transition system is intuitively described as follows: the model starts in some initial state $\sigma_0 \in I$ and evolves according to the transition relation $\rightarrowtriangle$, which means that if $\sigma \in \Sigma$ is the current state then a transition $(\sigma, \sigma') \in \rightarrowtriangle$ originating from σ is selected *nondeterministically*, then this selection procedure is repeated.

We write $\sigma \rightarrowtriangle \sigma'$ instead of $(\sigma, \sigma') \in \rightarrowtriangle$, and we write $\sigma \rightarrowtriangle$ if there exists $\sigma' \in \Sigma$ such that $\sigma \rightarrowtriangle \sigma'$. If there is no $\sigma' \in \Sigma$ such that $\sigma \rightarrowtriangle \sigma'$ we write $\sigma \not\rightarrowtriangle$.

In some examples, to ease the notation, if the set of states that identify a property is a singleton we will denote the state with the property it satisfies.

Remark 1.2. We are dealing with *unlabelled* transition systems in the sense that we are following a *state-based* to the description, as opposed to an *action-based* formulation. This means that we are abstracting from actions, and we are only interested in properties of the states, and we can map each state $\sigma \in \Sigma$ to the subset $\mathbf{P}_\sigma \subseteq \mathbf{P}$ of the properties it satisfies.

On the other hand an action-based approach would abstract from properties of states and refer to the action labels of the transitions; this approach is necessary to model communication.

Definition 1.3 (Successors, Predecessors)**.** Let $\sigma \in \Sigma$. We define the set of **direct successors** of σ as

$$\mathrm{Post}(s) = \{\sigma' \mid \sigma \rightarrowtriangle \sigma'\}$$

We define the set of **direct predecessors** of σ as

$$\mathrm{Pre}(s) = \{\sigma' \mid \sigma' \rightarrowtriangle \sigma\}$$

Definition 1.4 (Star)**.** Given a set X and $f : X \to X$, we define $f^* : X \to X$ on any $x \in X$ as

$$f^*(x) = \bigcup_{n \in \mathbb{N}} \{f^n(x)\}$$

Definition 1.5 (Reachable states)**.** The set of **reachable states from** σ, where $\sigma \in \Sigma$, can be formulated in terms of successors as

$$\sigma^{\rightarrowtriangle^*} = \text{Post}^*(\sigma) = \{\sigma' \mid \sigma \rightarrowtriangle^* \sigma'\}$$

where $\rightarrowtriangle^*$ is the reflexive and transitive closure of the relation $\rightarrowtriangle$.

Remark 1.6. We remark that there exist many classes of transition systems, suitable for different uses. A transition system together with the set of atomic proposition and with *non-blocking condition* (i.e. there are no dead-end states) is usually called a *Kripke Structure* [20].

In the following we consider only models in which all states always have an outgoing transition, i.e. for all $\sigma \in \Sigma$ there exists $\sigma' \in \Sigma$ such that $\sigma \rightarrowtriangle \sigma'$, since dead-end states (i.e. states with no outgoing transitions) can be encoded by adding to the model a dead state which loops on itself.

Definition 1.7 (Path)**.** A **path** (or *execution*) is an infinite sequence of states $\pi = \sigma_0 \to \sigma_1 \to \cdots \to \sigma_n \to \ldots$. We denote by $\pi_i = \sigma_i$ the i-th state of π, by π^k the path $\sigma_k \rightarrowtriangle \cdots \rightarrowtriangle \sigma_{k+n} \rightarrowtriangle \cdots$ for $k \in \mathbb{N}$. by $\pi[\sigma]$ the path starting at $\pi_0 = \sigma$, by $\pi(\Sigma) = \{\pi_i \mid i \in \mathbb{N}\}$ the set of states traversed by π.

Definition 1.8 (Model checking problem)**.** [18, Def. 2.2] Given a Kripke structure M with states $\sigma \in \Sigma$ and a specification φ in a temporal logic, the **Model Checking problem** is the problem of finding all the states $\overline{\sigma}$ such that $\overline{\sigma} \models \varphi$ and checking if the initial states are among this.

An *explicit state* model checker is a program which performs Model Checking directly on a Kripke structure.

Remark 1.9. Since we only consider finite state models, any infinite sequence $\pi = \sigma_0 \rightarrowtriangle \cdots \rightarrowtriangle \sigma_n \rightarrowtriangle \cdots$ can only traverse a finite number of different states. Let $\min_\pi \in \mathbb{N}$ denote the smallest index such that for any $k > \min_\pi$ we have $\sigma_k = \sigma_j$ for some $0 \leq j \leq \min_\pi$. It follows that $\pi(\Sigma) = \{\pi_i \mid i \in [0, \min_\pi]\}$ is determined by the finite prefix $\sigma_0 \rightarrowtriangle \cdots \rightarrowtriangle \sigma_{\min_\pi}$ of π and that any (state) property that is required to hold for all states in $\pi(\Sigma)$ can be decided over such finite prefix $\sigma_0 \rightarrowtriangle \cdots \rightarrowtriangle \sigma_{\min_\pi}$.

1.1.2 Properties and Temporal Logics

We briefly introduce the properties that are verified with Model Checking, and the main temporal logics used to specify them.

Safety and liveness Properties. A *requirement* for a component is a specification of acceptable or desired sequences of outputs in response to inputs, which should be stated as precisely as possible, and we may classify requirements into two categories: *safety* requirements assert that "nothing bad ever happens", while *liveness* requirements assert that "something good eventually happens". Such requirements are specified using a formalism called *temporal logic*, and the problem of checking whether a model satisfies its specification expressed in temporal logic is known as *Model Checking*.

The verification problem consists in checking whether the given implementation meets these requirements: a *violation* of a *safety* requirement can be a finite execution that illustrates the undesirable behaviour, i.e. a *counterexample*. On the other hand *no finite* execution can demonstrate the violation of a liveness property: a *witness* should show a *cycle* in which the system gets stuck without achieving the goal.

A typical example of safety property is the mutual exclusion property, i.e. the fact that always at most one process is in its critical section; another typical safety property is the absence of deadlocks, i.e. of (reachable) states with no exiting transitions.

Safety properties are requirements of a particular kind: they are *invariants*, i.e. properties that hold for all reachable states. The dual concept of invariant property is *reachable property*: a property is reachable if it is satisfied by some reachable state of the transition system. It follows that a property φ of a transition system is an invariant if and only if the negated property $\neg\varphi$ is not reachable. This idea is often exploited to verify safety requirements.

An algorithm could easily fulfill a safety property by doing nothing, since doing nothing does not lead to "bad" states. But this is not the behaviour we look for in a system that should *do* something. Thus, safety properties are complemented by liveness properties, i.e. properties that require that some good behaviour occurs. An example of liveness property is the request that some output corresponds to an input, or that a request for a resource eventually gets granted.

Linear-time and branching-time Properties. Another axis along which we could classify the properties we are interested is the linear-time vs. branching-time properties. Linear-time properties specify the traces that a transition system should exhibit, i.e. it is a requirement on the states traversed by those traces of the transition system. Linear time properties thus focus on the behaviour of a single execution path through the state space: in the *linear* view of time, at each moment in time there is a single successor moment; in the *branching* view of time, on the other hand, the structure can be branching, tree-like: time can split in alternative courses. Thus, branching-time properties focus on considering multiple possible execution paths or behaviours simultaneously: this is especially useful when dealing with non-deterministic or concurrent systems, where different executions could lead to different outcomes.

Linear Temporal Logic, for short LTL, is a linear-time modal logic that expresses linear-time properties, while Computation Tree Logic, for short CTL, is a branching-time modal logic that focuses on branching-time properties. The expressiveness of these two logics is not comparable, i.e. there are LTL formulae which cannot be expressed in CTL, and vice versa. Both are subsumed by the branching-time logic CTL*, whose expressive power is in turn contained in that of μ-calculus.

Temporal Modalities. Temporal logics such as LTL and CTL are propositional logics to which *temporal modalities* are added to express properties that allow for the specification of the *relative order* of events. Temporal modalities are thus expressions used to describe and reason about properties that involve time and ordering of events, formally capturing temporal relationships between states or events in a system's behaviour. We briefly describe the most common temporal operators (they are not all fundamental, since some of them can be derived from the others):

- the *next* modality, denoted by $\bigcirc$ or X, is used to express that a certain condition is true in the next step along a path;

- the *eventually* modality, denoted by $\Diamond$ or F, is used to express that a certain condition will eventually become true along a path;

- the *always* modality, denoted by $\Box$ or G, is used to express that a certain condition remains true for all steps along a path;

- the *until modality*, denoted by $\mathcal{U}$ or U, is used to express that a certain condition holds until another condition becomes true at some point in the future along a path.

In CTL two other modalities will be added to quantify over paths: the A modality quantifies universally over paths, while the E modality quantifies existentially over paths, but they are constrained to be immediately followed by a temporal operator. In CTL*, a propositional temporal logic that contains both as subfragments, and that we introduce here following [26], these constraints are removed.

CTL*. CTL* is a propositional modal logic with *path quantifiers* (A, E), which are interpreted over states, and *temporal operators* $(\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U})$, which are interpreted over paths.

Definition 1.10 (CTL* syntax)**.** Given a set of atomic propositions $\mathbf{P}$ with $\mathsf{p} \in \mathbf{P}$, the syntax of CTL* is recursively defined as follows:

$$\varphi \ ::= \mathsf{p} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{A}\ \varphi \mid \mathsf{E}\ \varphi \mid \mathsf{X}\ \varphi \mid \mathsf{F}\ \varphi \mid \mathsf{G}\ \varphi \mid \varphi_1\ \mathsf{U}\ \varphi_2$$

In particular, we can distinguish two synctactic subsets of CTL*: *state formulae* are boolean combinations of atomic propositions and CTL* formulae whose outermost operator is a path quantifier, while *path formulae* are those whose outermost operator is a temporal operator. The truth value of a state formula can be asserted over a state in a Kripke structure, while we need a path to determine the truth value of a path formula.

Not all of these operators are really necessary: it can be shown that the full expressive power of CTL* can be obtained by just using X, U and one among A, E. We now describe the semantics for CTL*.

Definition 1.11 (CTL* semantics)**.** Given a Kripke structure $(\Sigma, I, \mathbf{P}, \twoheadrightarrow, \vdash)$ let $\sigma \in \Sigma$ be a state, π be a path, $\mathsf{p} \in \mathbf{P}$ an atomic proposition, f, g state formulae, φ and ψ CTL* formulae, we have:

$$
\begin{array}{rll}
\sigma \models \mathsf{p} & \text{iff} & \sigma \vdash \mathsf{p} \\
\sigma \models \neg f & \text{iff} & \sigma \not\models f \\
\sigma \models f \vee g & \text{iff} & \sigma \models f \text{ or } \sigma \models g \\
\sigma \models f \wedge g & \text{iff} & \sigma \models f \text{ and } \sigma \models g \\
\sigma \models \mathsf{E}\ \varphi & \text{iff} & \text{there is an infinite path } \pi \text{ such that } \pi_0 = \sigma \text{ and } \pi \models \varphi \\
\sigma \models \mathsf{A}\ \varphi & \text{iff} & \text{for every infinite path } \pi \text{ such that } \pi_0 = \sigma \text{ we have } \pi \models \varphi \\
\pi \models f & \text{iff} & \pi_0 \models f \\
\pi \models \neg\varphi & \text{iff} & \pi \not\models \varphi \\
\pi \models \varphi \vee \psi & \text{iff} & \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models \varphi \wedge \psi & \text{iff} & \pi \models \varphi \text{ and } \pi \models \psi \\
\pi \models \mathsf{X}\ \varphi & \text{iff} & \pi^1 \models \varphi \\
\pi \models \mathsf{F}\ \varphi & \text{iff} & \text{there exists an } i \geq 0 \text{ such that } \pi^i \models \varphi \\
\pi \models \mathsf{G}\ \varphi & \text{iff} & \text{for all } j \geq 0 \text{ we have } \pi^j \models \varphi \\
\pi \models \varphi\ \mathsf{U}\ \psi & \text{iff} & \text{there exists an } j \geq 0 \text{ such that } \pi^j \models \psi \text{ and for all } 0 \leq i < j \text{ we have } \pi^i \models \varphi
\end{array}
$$

LTL. LTL is a syntactic fragment of CTL* for the description of linear-time properties. LTL formulae over the set $\mathbf{P}$ of atomic propositions can be formed according to the following grammar, in which no path quantifier is allowed except for a leading A.

Definition 1.12 (LTL syntax)**.** Given a set of atomic propositions $\mathbf{P}$ with $\mathsf{p} \in \mathbf{P}$, let φ range over LTL$^-$ formulae, and build LTL formulae as $\mathsf{A}\ \varphi$:

$$\varphi \ ::= \mathsf{p} \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\ \varphi \mid \mathsf{F}\ \varphi \mid \mathsf{G}\ \varphi \mid \varphi_1\ \mathsf{U}\ \varphi_2$$

LTL is the *linear-time* fragment of CTL* because LTL$^-$ formulae are interpreted over paths, i.e. linear sequences of states.

Paths are obtained from a transition system that might be branching: a state may have distinct direct successor states, so different computations can branch from the same state. The interpretation of LTL formulae in a state requires that a formula φ in a state s holds if *all* possible computation that start in s satisfy φ.

CTL. CTL is a synctactic fragment of temporal logic to describe branching-time properties. In CTL every path quantifier must immediately be followed by a temporal operator.

Definition 1.13 (CTL syntax)**.** Given a set of atomic propositions $\mathbf{P}$ with $\mathsf{p} \in \mathbf{P}$, the syntax of CTL is recursively defined as follows:

$$\varphi ::= \mathsf{p} \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{AX}\ \varphi \mid \mathsf{EX}\ \varphi \mid \mathsf{AF}\ \varphi \mid \mathsf{EF}\ \varphi \mid \mathsf{AG}\ \varphi \mid \mathsf{EG}\ \varphi \mid \mathsf{A}\ \varphi_1\ \mathsf{U}\ \varphi_2 \mid \mathsf{E}\ \varphi_1\ \mathsf{U}\ \varphi_2$$

Every CTL formula, hence every subformula of a CTL formula, is a state formula.

We introduce a little example taken from [15], which we will discuss later with more details in 3.11, to underline the different expressive power of LTL and ACTL.

Example 1.14. [15, Ex. 1.2] Consider the transition system in Figure 1.1, with $\mathbf{P} = \{\mathsf{tt}, \mathsf{a}, \neg\mathsf{a}, \mathsf{ff}\}$, where $\sigma_0, \sigma_2 \vdash \mathsf{a}$, $\sigma_1 \not\vdash \mathsf{a}$. We can see that the formulae $\mathsf{AFG}\ \mathsf{a}$ and $\mathsf{AF}\ \mathsf{AG}\ \mathsf{a}$ are not equivalent: the first formula holds on this transition system (either the system loops forever in σ_0, in which a holds, or, if at a certain point there is a transition to σ_1, then there is a transition to σ_2 and the system loops forever in σ_2, in which again a holds), while the second formula does not hold, since $\sigma_0 \not\models \mathsf{AG}\ \mathsf{a}$ and the system could loop in σ_0.

$$\begin{tikzcd}
{} \arrow[d] & & \\
\sigma_0 \arrow[r] \arrow[loop left] & \sigma_1 \arrow[r] & \sigma_2 \arrow[loop right]
\end{tikzcd}$$

Figure 1.1: An example to distinguish AFG and $\mathsf{AF}\ \mathsf{AG}$.

μ-calculus. μ-calculus is a more expressive logic that subsumes CTL* and all its synctactic fragments. μ-calculus is a modal logic describing properties of transition systems, on which we require a countable set of variables, whose meaning will be set of states; μ-calculus allows for the explicit description of fixed-point properties by extending a propositional modal logic with the *least fixed-point* operator μ and the *greatest fixed-point* operator ν. Its semantics can be characterised in terms of parity games.

To summarise, we copy here a diagram from [26] that illustrates the basic temporal logic we have discussed and the ones we will be interested in, pointing out their relations:

$$\begin{tikzcd}
 & \mu\text{-calculus} & & \\
 & \text{CTL}^* \arrow[u] & & \\
\text{CTL} \arrow[ur] & & \text{ACTL}^* \arrow[ul] & \\
 & \text{ACTL} \arrow[ul] \arrow[ur] & & \text{LTL} \arrow[ul]
\end{tikzcd}$$

In particular, CEGAR (Sec. 1.2.2) works on ACTL*, while our focus will be devoted to ACTL.

1.2 ACTL

ACTL formulae are temporal formulae universally quantified over all paths leaving the current state, thus ACTL is the fragment of CTL in which only the A operator is allowed. Counterexamples to universal properties are existential, hence a single failure path is sufficient to show that a certain property does not hold; however the form of counterexamples could be made more complicated by the nesting of properties.

1.2.1 Syntax and Semantics

Definition 1.15 (ACTL syntax)**.** Given a set of atomic propositions $\mathbf{P}$ with $\mathsf{p} \in \mathbf{P}$, the syntax of ACTL is recursively defined as follows:

$$\varphi \ ::= \ \mathsf{p} \mid \neg\mathsf{p} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{AX}\ \varphi_1 \mid \mathsf{AF}\ \varphi_1 \mid \mathsf{AG}\ \varphi_1 \mid \varphi_1\ \mathsf{AU}\ \varphi_2$$

The **satisfaction relation** is obtained by instantiating the general notion given for CTL in Definition 1.13.

Definition 1.16. The **satisfaction relation** for ACTL formulae over *states* can be defined by structural induction as follows:

$$\begin{array}{rcl}
\sigma \models \mathsf{p} & \text{iff} & \sigma \vdash \mathsf{p} \\
\sigma \models \neg\mathsf{p} & \text{iff} & \sigma \vdash \neg\mathsf{p} \\
\sigma \models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ and } \sigma \models \varphi_2 \\
\sigma \models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma \models \varphi_1 \text{ or } \sigma \models \varphi_2 \\
\sigma \models \mathsf{AX}\ \ \varphi_1 & \text{iff} & \forall \sigma \twoheadrightarrow \sigma'.\ \sigma' \models \varphi_1 \\
\sigma \models \mathsf{AF}\ \ \varphi_1 & \text{iff} & \forall \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n \twoheadrightarrow \cdots.\ \exists k \in \mathbb{N}.\ \sigma_k \models \varphi_1 \\
\sigma \models \mathsf{AG}\ \ \varphi_1 & \text{iff} & \forall \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n \twoheadrightarrow \cdots.\ \forall j \in \mathbb{N}.\ \sigma_j \models \varphi_1 \\
 & \text{iff} & \forall \sigma \twoheadrightarrow^* \sigma'.\ \sigma' \models \varphi_1 \\
\sigma \models \varphi_1\ \ \mathsf{AU}\ \ \varphi_2 & \text{iff} & \forall \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n \twoheadrightarrow \cdots. \\
 & & \exists k \in \mathbb{N}.\ (\sigma_k \models \varphi_2 \text{ and } \forall j \in [0, k-1].\ \sigma_j \models \varphi_1)
\end{array}$$

Remark 1.17. Note that AF is definable in terms of AU as it can be readily checked that $\mathsf{AF}\ \varphi = \mathsf{tt}\ \mathsf{AU}\ \varphi$.

1.2.2 CEGAR

Counterexample-Guided Abstraction Refinement, (CEGAR) is a popular abstraction refinement technique applicable for the verification of temporal logic formulae. In the original paper [18] the framework is used to verify properties specified in ACTL*: this framework gives tools to combine an automatic generation of an initial abstraction, a symbolic Model Checking procedure and an automatic refinement of the abstraction based on counterexamples, in an iterative way.

Symbolic Model Checking. In Symbolic Model Checking the transition relation of a Kripke structure is not explicitly constructed, but instead a Boolean function is computed to represent it. Then fixed point characterizations of temporal operators are applied to the Boolean function rather then to the Kripke structure. These Boolean functions are represented in a compact and efficient way as *ordered binary decision diagrams*, BDDs. The BDDs data structure is particularly popular in Symbolic MC due to its ability of representing and manipulate large sets of states or complex Boolean formula. A Symbolic Model Checking algorithm is an algorithm in which variables do not denote single states, but set of states are represented by Boolean functions.

Definition 1.18 (Binary Decision Diagram)**.** A **Binary Decision Diagram** is a directed acyclic graph with two types of vertices: terminal and internal. *Terminal vertices* have no outgoing edges and are labelled with a Boolean constant (0 or 1). *Internal Vertices* are labelled with a variable $x \in X$ and have two outgoing edges: a *right* one (which represents setting the variable to 1) and a *left* one (which represents setting the variable to 0).

In the previous definition, every path from an internal vertex to a terminal vertex contains at most one vertex for each variable $x \in X$: each vertex u represents a Boolean function $f(u)$, and given a valuation for X the value of $f(u)$ is obtained by traversing the path starting from u and choosing at each vertex the left or right edge based on the value assigned in the specific valuation to the variable it represents. The whole idea of Binary Decision Diagrams is based on *Shannon's expansion formula.*

Definition 1.19 (Shannon's expansion formula)**.** Given a set of k Boolean variables X and a Boolean formula f over X, which can be represented as $f : \texttt{bool}^k \to \texttt{bool}$, we have that the **Shannon expansion of f over the variable $x \in X$** is

$$f \cong (\neg x \wedge f[x \mapsto 0]) \vee (x \wedge f[x \mapsto 1])$$

Where $f[x \mapsto b]$ for $b = 0, 1$ are obtained from f by substituting x with the value b.

The formulae resulting from expanding on the variable x do not refer to x anymore, hence they are simpler Boolean functions, depending on one less variable. We can apply Shannon's formula recursively to simplify Boolean functions, leading to a representation as *decision diagrams.*

Abstraction. The use of abstraction allows us to tackle the main challenge in Model Checking, that is state explosion. An *existential abstraction* is a partitioning of states of a Kripke structure into clusters, which are treated as new abstract state. The abstraction function may be defined as a surjective function from states to abstract states, and a new Kripke structure may be induced from the original one and the abstraction. The abstraction should be *appropriate* with respect to the specification φ, i.e. if two concrete states are identified then they must share the same truth value for all subformulae of φ. This leads to the notion of *consistent* abstract state (collapsing a set of concrete states into an abstract state does not lead to contradictions in the properties satisfied by an abstract state).

The initial abstraction function is obtained by identifying those concrete states that cannot be distinguished by the atomic formulae contained in φ.

The key step of CEGAR is to extract information from spurious counterexample, i.e. false negatives due to over-approximation. As we said, over-approximation is sound, i.e. if a specification is true in the abstract model, then it is true also in the concrete design, but it the specification is false, the counterexample may be the result of some behaviour that was introduced with the approximation, and is not present in the original model. This method is also *complete*, meaning that no false negatives are found, for an important fragment of the ACTL* temporal logic: in particular, the CEGAR paper only refers to safety property and counterexamples which can be represented as finite or infinite *linear paths.*

Remark 1.20. As pointed out in [15], counterexamples are not all expressible as linear paths, some necessarily have a tree-like, branching structure. The paper gives precise definitions to *multi-paths* and *linear-paths* and identifies a set of templates of ACTL for which a linear-path exists, and it proves that it is in general NP-hard to produce complete counterpaths. No characterization exists (to our knowledge) of the biggest fragment of ACTL for which linear counterexamples exist.

Procedure. The main steps of the iterative procedure can be described as follows:

1. given a program P with a corresponding Kripke structure M, generate an initial abstraction h by analyzing the transition blocks corresponding to the variables of the program. Let $\widehat{M}$ be the abstract Kripke structure corresponding to h;
2. model-check the abstract structure $\widehat{M}$:
   - if φ holds for $\widehat{M}$, we can conclude it also holds for M and we are done;
   - if instead a counterexample $\hat{T}$ is found, we must check if it is a real or a spurious counterexample. Since the model is finite, this step can be done by simulating $\hat{T}$ on the actual model. If it is an actual counterexample it is reported, else the procedure goes to step 3;
   - the abstraction is refined by partitioning a single equivalence class so that after the refinement the abstract structure $\widehat{M}$ does not admit the spurious counterexample $\hat{T}$. After the refinement, the procedure goes back to step 2.

We discuss an easy example from the original CEGAR paper [18] to show an example of an incomplete analysis and one of a complete analysis. This example will be reprised in Ex. 4.23 to illustrate our abstractions techniques.

Example 1.21. [18, Ex. 3.4, 3.7] Consider the transition system in Figure 1.2, which represents a US traffic light controller.

$$\begin{tikzcd} {} \arrow[r] & \mathsf{red} \arrow[r] & \mathsf{green} \arrow[r] & \mathsf{yellow} \arrow[ll, bend left] \end{tikzcd}$$

Figure 1.2: Traffic light example from [18].

We want to prove that the property $\varphi = \mathsf{AG\ AF\ red}$ holds using the abstraction function

$$h : \mathsf{red} \mapsto \widehat{\mathsf{red}} \qquad h : \mathsf{green}, \mathsf{yellow} \mapsto \widehat{\mathsf{go}}$$

The abstract transition system is depicted in Figure 1.3. We can see that the property holds for the original system, but it does not hold for the abstract system, since there is an infinite abstract trace $(\widehat{\mathsf{red}}, \widehat{\mathsf{go}}, \widehat{\mathsf{go}}, \dots)$ that invalidates the property: this trace is a spurious counterexample.

We may now consider a variation of this transition system, in which we model also the behaviour of a car in reaction to the behaviour of the traffic light (the drive state of the car is guarded by the green state of the traffic light). The two transition systems and their composition are represented in Figure 1.4.

$$\begin{tikzcd}
{} \arrow[r] & \widehat{\mathsf{red}} \arrow[r, bend left] & \widehat{\mathsf{go}} \arrow[l, bend left] \arrow[loop right]
\end{tikzcd}$$

Figure 1.3: Abstract transition system for the traffic light.

$$\begin{tikzcd}
\mathsf{red} \arrow[r] & \mathsf{green} \arrow[r] & \mathsf{yellow} \arrow[ll, bend left]
\end{tikzcd}
\qquad
\begin{tikzcd}
{} \arrow[d] \\
\mathsf{stop} \arrow[loop right] \arrow[d, shift left, "\mathsf{green}"] \\
\mathsf{drive} \arrow[u, shift left] \arrow[loop right, "\mathsf{green}"]
\end{tikzcd}$$

$$\begin{tikzcd}
{} \arrow[r] & \mathsf{rs} \arrow[loop above] \arrow[r] & \mathsf{gs} \arrow[loop above] \arrow[r] \arrow[d, shift right] \arrow[dr] & \mathsf{ys} \arrow[loop above] \arrow[ll, bend right] \\
& \mathsf{rd} \arrow[u] \arrow[ur] & \mathsf{gd} \arrow[u, shift right] \arrow[ur] \arrow[r] \arrow[loop below] & \mathsf{yd} \arrow[u] \arrow[ull]
\end{tikzcd}$$

Figure 1.4: Composite transition system of traffic light and car.

We want to prove that the safety property $\varphi' = \mathsf{AG}\ (\neg\mathsf{rd})$ is true for the system, and we see that we can do so by applying the abstraction h defined above to the composite transition system, which is represented in Figure 1.5, since the state $\hat{\mathsf{r}}\mathsf{d}$ is unreachable (as is the concrete state rd).

$$\begin{tikzcd}
{} \arrow[r] & \hat{\mathsf{r}}\mathsf{s} \arrow[loop above] \arrow[r, bend left] & \hat{\mathsf{g}}\mathsf{s} \arrow[loop above] \arrow[l, bend left] \arrow[d, shift left] \\
& \hat{\mathsf{r}}\mathsf{d} \arrow[u] \arrow[ur] & \hat{\mathsf{g}}\mathsf{d} \arrow[u, shift left] \arrow[ul] \arrow[loop below]
\end{tikzcd}$$

Figure 1.5: Abstract transition system for the composition of traffic light and car.

Chapter 2

Abstract Interpretation

Abstract Interpretation was first introduced by Patrick and Radhia Cousot in [2, 3] as a sound-by-construction method for verification in static analysis, which is aimed to discuss properties of programs, given a model of the program's behaviour. It is the theory of sound approximation of the semantics of programs, based on monotonic functions over ordered sets, especially lattices. The core idea of Abstract Interpretation is to create an abstract domain that captures the essential aspects of a program behaviour, for example reasoning about the program's variables and their relationships, while ignoring details that would make the analysis computationally infeasible. For an analysis of the history of Abstract Interpretation, see [31].

This technique carries the issue of completeness, which is closely related to the goal of deriving the most abstract domain to decide program correctness without raising false alarms. Completeness intuitively encodes the greatest achievable precision for a program analysis on a given abstract domain, but unfortunately the most abstract refinement, which comes as solution of a recursive domain equation, as currently known yields an abstract domain that is often way too fine grained.

2.1 General concepts

2.1.1 Order theory

Partial orders are extremely important in several areas of Theoretical Computer Science, especially in the field of semantics. As pointed out in [24], in Abstract Interpretation partial orders are used at different levels of the theory to model core notions. First of all, partial orders convey the idea of *approximation*: some analysis results may be coarser than some other results, and the order is partial as sometimes analysis results are incomparable. Secondly, partial orders convey the idea of *validating a specification*: we say that a program P satisfies a specification Spec if $[\![P]\!] \subseteq \text{Spec}$, meaning that the program behaviours are contained in the set of admissible behaviours. Moreover, partial orders convey the idea of *soundness*: we say that an analysis is sound its result is coarser than the actual behaviour.

Thus we briefly introduce the mathematical notions that are necessary to discuss the Abstract Interpretation formalism, namely order and lattice theory.

Definition 2.1 (Partially ordered set)**.** Let X be a non-empty set. We say that $(X, \leq_X)$ is a **partially ordered set** if $\leq_X \subseteq X \times X$ is a reflexive, antisymmetric and transitive relation, i.e. for all $x, y, z \in X$

- $x \leq_X x$ (reflexivity);
- if $x \leq_X y$ and $y \leq_X x$ then $x = y$ (antisymmetry);
- if $x \leq_X y$ and $y \leq_X z$ then $x \leq_X z$ (transitivity).

Definition 2.2 (Least upper bound)**.** Let $(X, \leq_X)$ be a partially ordered set and let $Z \subseteq X$. We say that $\overline{z}$ is an **upper bound** on Z if $z' \leq_X \overline{z}$ for all $z' \in Z$. It is a **least upper bound** of Z, denoted $\vee_X Z$ if it is an upper bound on Z and if z is another upper bound on Z then we have $\overline{z} \leq_X z$.

We may give the dual definition to least upper bound, obtained by reversing the inequalities:

Definition 2.3 (Greatest lower bound)**.** Let $(X, \leq_X)$ be a partially ordered set and let $Z \subseteq X$. We say that $\overline{z}$ is a **lower bound** on Z if $\overline{z} \leq_X z'$ for all $z' \in Z$. It is a **greatest lower bound** on Z, denoted $\wedge_X Z$, if it is a lower bound on Z and if z is another lower bound on Z then we have $z \leq_X \overline{z}$.

When the set X is clear from the context, we omit to indicate it and just write $\vee, \wedge$. It can be easily seen that if Z has a least upper bound then it is indeed unique. Dually if Z has a greatest lower bound.

In Abstract Interpretation we usually work in domains in which the existence of upper and lower bounds is ensured for every possible subset: these domains are called complete lattices.

Definition 2.4 (Complete Lattice)**.** A partially ordered set $(X, \leq_X)$ is a **complete lattice** if every subset of X has both least upper bound and greatest lower bound.

A useful concept for abstract domains is Moore closure, which ensures precision in representation by guaranteeing that the abstract domain contains all possible program evaluations in a comprehensive manner and avoids the loss of relevant information. This notion is particularly valuable in the repair of abstract domains.

Definition 2.5 (Moore closure)**.** Given a complete lattice $(X, \leq_X)$ and $Z \subseteq X$ the **Moore closure** of Z is defined as

$$\mathcal{M}(Z) = \{\wedge_X Y \mid Y \subseteq Z\}$$

which means that $\mathcal{M}(Z)$ is the least superset of X closed under greatest lower bounds of its subsets.

We provide some concepts related to functions that will be useful in the description of Galois connections, closures, and their properties.

Remark 2.6. Let X be a set, let $(Y, \leq_Y)$ be a partially ordered set and let $f, g : X \to Y$. We denote by $f \sqsubseteq g$ the *point-wise ordering*, i.e.

$$f \sqsubseteq g \quad \Leftrightarrow \quad f(x) \leq_Y g(x) \; \forall x \in X$$

Definition 2.7 (Extensive, reductive, idempotent function)**.** Given a complete lattice $(X, \leq_X)$ and a function $g : X \to X$ we say that g is **extensive** if $\mathrm{id}_X \sqsubseteq g$, we say that g is **reductive** if $g \sqsubseteq \mathrm{id}_X$, and we say that it is **idempotent** if $g \circ g = g$.

Definition 2.8 (Monotone, additive, co-additive function)**.** Given two complete lattices $(X, \leq_X)$ and $(Y, \leq_Y)$, and a function $f : X \to Y$ we say that f is **monotone** if $x_1 \leq_X x_2$ implies $f(x_1) \leq_Y f(x_2)$.

We say that f is **additive** if it preserves arbitrary least upper bounds, **coadditive** if it preserves arbitrary greatest lower bounds.

2.1.2 Abstract Interpretation

We now give precise definitions to the notion of *approximation* and *abstraction* in the framework of Abstract Interpretation in two different but equivalent ways: we first introduce *Galois connections* as a formalism that captures the correspondence between the concrete and the abstract domain by means of an abstraction map and a concretization map. Galois connections stem from Galois theory, that studies the solvability of polynomial equations; they have applications in several branches of Mathematics and they prominently appear in Computer Science as basic tool for Abstract Interpretation.

Then we introduce *closure operators*, which are maps that are monotone, extensive and idempotent. In this framework we interpret a closure operator on an ordered algebraic structure of properties of program state, and the result of applying the closure operator to all the properties of interest on the program behaviour is the abstract domain.

Definition 2.9 (Galois connection)**.** A **Galois connection** is a tuple (α, C, A, γ) where

- A, C are complete lattices;
- $\alpha : C \to A$ and $\gamma : A \to C$ are monotone;
- for all $c \in C$ and for all $a \in A$ we have $c \leq_C \gamma(a)$ if and only if $\alpha(c) \leq_A a$.

A is called the *abstract domain*, C is called the *concrete domain*, α is called the *abstraction map*, γ is called the *concretization map*.

$$\begin{tikzcd} A \arrow[d, bend left, "\gamma"] \\ C \arrow[u, bend left, "\alpha"] \end{tikzcd}$$

The order on A, C encodes a precision (approximation) relation: smaller means more precise. The element $\alpha(c)$ is the best approximation of c on A, i.e. it is the smallest abstract element which over-approximates c: formally, $\alpha(c) = \bigwedge \{a \mid c \leq_C \gamma(a)\}$.

Definition 2.10 (Expressible element)**.** Let $c \in C$ be an element of the concrete domain. We say that c is **expressible** in the abstract domain $A_{\alpha,\gamma}$ if $A(c) = c$.

Remark 2.11. If (α, C, A, γ) is a Galois connection, then α is additive, γ is co-additive, and $\gamma(A) \subseteq C$ is Moore closed. Moreover, $\alpha \circ \gamma$ is reductive, i.e. $\alpha \circ \gamma \sqsubseteq \mathrm{id}_A$, and $\gamma \circ \alpha$ is extensive, i.e. $\gamma \circ \alpha \sqsupseteq \mathrm{id}_C$.

Definition 2.12 (Galois insertion)**.** A Galois connection (α, C, A, γ) is a **Galois insertion** if one of the following equivalent conditions holds:

1. α is surjective;
2. γ is injective;
3. for all $a \in A$ we have $\alpha(\gamma(a)) = a$

The intuition for Galois insertions is that they eliminate useless abstract values: the fact that α should be surjective represents the fact that we do not need abstract elements that do not represent any concrete value; the fact that γ should be injective represents the fact that it would be meaningless to have two identical abstract values.

Definition 2.13 (Class of Abstract Domains on C)**.** The **class of abstract domains on** C is given by

$$\text{Abs}(C) = \{A_{\alpha,\gamma} \mid (\alpha, C, A, \gamma) \text{ is a GI}\}$$

We note that $(\text{Abs}(C), \sqsubseteq)$ is actually a complete lattice, where $A' \sqsubseteq A$ means that A' is a refinement of A (i.e. it is more precise).

Definition 2.14 (Closure operator)**.** Given a complete lattice $(X, \leq_X)$ and a function $\mu : X \to X$ we say that g is a **closure operator** if it is monotone, idempotent and extensive.

The fact that the two formulations are equivalent is made explicit by the following well-known proposition, which allows us to equivalently define abstract domains by means of the Galois insertion definition, or by giving a closure operator.

Proposition 2.15. *If $A_{\alpha,\gamma} \in \text{Abs}(C)$ is a Galois insertion then $\gamma \circ \alpha$ is a closure operator, and if μ is a closure operator on C then $\mu(C)_{\mu,\text{id}} \in \text{Abs}(C)$ is a Galois insertion.*

In order to perform calculations in the abstract domain, we need to also define the corresponding abstract versions of operations in the concrete domain and provide conditions under which abstraction is a correct approximation of the underlying operation. The best correct approximation of a function f is the smallest correct abstraction of f.

Definition 2.16 (Correct, complete approximation)**.** Given an abstract domain $A_{\alpha,\gamma} \in \text{Abs}(C)$ and a function $f : C \to C$, an abstract function $f^\sharp : A \to A$ is a **correct approximation** of f if $\alpha \circ f \sqsubseteq f^\sharp \circ \alpha$ holds. An abstract function $f^\sharp : A \to A$ is a **complete approximation** of of f if $\alpha \circ f = f^\sharp \circ \alpha$ holds.

Definition 2.17 (Best correct approximation)**.** Given an abstract domain $A_{\alpha,\gamma} \in \text{Abs}(C)$ and a function $f : C \to C$, the **best correct approximation**, for short BCA, f^A of f in A is defined as

$$f^A = \alpha \circ f \circ \gamma : A \to A$$

Definition 2.18 (Completeness)**.** An abstract domain $A \in \text{Abs}(C)$ is **complete** for a function $f : C \to C$, denoted by $\mathbb{C}^A(f)$, if $A \circ f = A \circ f \circ A$.

2.1.3 Kleene language

Following [28, 32] we introduce a language of regular commands Reg, based on Kozen's Kleene algebra with test [10]. It is parametric on the syntax of basic expressions $\mathsf{e} \in \mathsf{Exp}$, which provide the basics command and can be instantiated with different instructions, qualifying the language we are interested in. In this first part we use it to model a deterministic imperative language to illustrate how the rules of the Local Completeness Logic work; later we will use it to encode the computation of ACTL counterexamples.

Definition 2.19 (Language of regular commands)**.** Let Exp be a collection of basic expressions and let $\mathsf{e} \in \mathsf{Exp}$. We define a language of regular commands Reg, parametric on Exp, as

$$\mathsf{Reg} \ni \mathsf{r} ::= \mathsf{e} \mid \mathsf{r_1}; \mathsf{r_2} \mid \mathsf{r_1} \oplus \mathsf{r_2} \mid \mathsf{r_1^*}$$

We briefly comment on the meaning of these expressions: the term $\mathsf{r_1}; \mathsf{r_2}$ represents sequential composition, the term $\mathsf{r_1} \oplus \mathsf{r_2}$ represents non-deterministic choice command, and the term $\mathsf{r_1^*}$ represents the Kleene iteration of $\mathsf{r_1}$, which means that $\mathsf{r_1}$ can be performed zero or any finite number of times.

Assuming that basic expressions have a concrete semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ on a complete lattice C, such that $(\!|\mathsf{e}|\!)$ is an additive function (this assumption can be done without loss of generality since the collecting semantics is defined by additive lifting), we can define the semantics of the language of regular expressions as follows.

Definition 2.20 (Concrete semantics of regular expressions)**.** The concrete semantics of regular expressions $[\![\cdot]\!] : \mathsf{Reg} \to C \to C$ is inductively defined by

$$\begin{aligned} [\![\mathsf{e}]\!]c &= (\!|\mathsf{e}|\!)\, c \\ [\![\mathsf{r_1}; \mathsf{r_2}]\!]c &= [\![\mathsf{r_2}]\!]([\![\mathsf{r_1}]\!]c) \\ [\![\mathsf{r_1} \oplus \mathsf{r_2}]\!]c &= [\![\mathsf{r_1}]\!]c \vee [\![\mathsf{r_2}]\!]c \\ [\![\mathsf{r_1^*}]\!]c &= \bigvee \{[\![\mathsf{r_1}]\!]^k c \mid k \in \mathbb{N}\} \end{aligned}$$

Remark 2.21. Observe that, if $[\![\cdot]\!]$ is additive, then $[\![\mathsf{r_1^*}]\!]c$ is the least fixed point of the function $\lambda x.[\![r_1]\!]x \vee c$. This could be relevant because the definition as fixed point is computationally easier to handle, but it is not relevant to our goals.

Building an imperative language We now instantiate the basic expressions as in [32]: the basic expressions considered are those used in deterministic while programs, namely no-operation instruction, assignments and Boolean guards.

Definition 2.22 (Syntax of basic expressions for the imperative language)**.** We define the syntax of basic expressions for the Imp language as

$$\mathsf{Exp} \ni \mathsf{e} ::= \mathsf{skip} \mid x := \mathsf{a} \mid \mathsf{b}?$$

where a is an arithmetic expression on integer values and variables $x \in \mathrm{Var}$, while b ranges over Boolean expressions, including negation.

Definition 2.23 (Syntax of basic expressions for the imperative language)**.** We define the semantics of basic expressions for the Imp language as

$$\begin{aligned} (\!|\mathsf{skip}|\!)\, p &= p \\ (\!|x := \mathsf{a}|\!)\, p &= \{\sigma[x \mapsto \{\!|\mathsf{a}|\!\}\,\sigma] \mid \sigma \in p\} \\ (\!|\mathsf{b}?|\!)\, p &= \{\sigma \in p \mid \{\!|\mathsf{b}|\!\}\,\sigma = \mathsf{tt}\} \end{aligned}$$

where $\sigma[x \mapsto v]$ is the store update, $\{\!|\mathsf{a}|\!\} : \Sigma \to \mathbb{Z}$ is the arithmetic expressions semantics and $\{\!|\mathsf{a}|\!\} : \Sigma \to \{\mathsf{tt}, \mathsf{ff}\}$ is the boolean expressions semantics.

Then we can introduce the if-then-else and loop commands as syntactic sugar as follows:

$$\text{if } \mathsf{b} \text{ then } \mathsf{c}_1 \text{ else } \mathsf{c}_2 = (\mathsf{b}?; \mathsf{c}_1) \oplus (\mathsf{b}?; \mathsf{c}_2)$$

$$\text{while } \mathsf{b} \text{ do } \mathsf{c} = (\mathsf{b}?; \mathsf{c})^*; \neg\mathsf{b}?$$

Definition 2.24 (Syntax of Imp)**.** We define the syntax of Imp commands with the following grammar:

$$\mathsf{Imp} \ni \mathsf{c} ::= \mathsf{skip} \mid x := \mathsf{a} \mid c; c \mid \text{if } \mathsf{b} \text{ then } \mathsf{c}_1 \text{ else } \mathsf{c}_2 \mid \text{while } \mathsf{b} \text{ do } \mathsf{c}$$

In this setting, a program store is $\sigma : V \to \mathbb{Z}$ a total function from a finite set of variables $V \subseteq \mathrm{Var}$ to values, and $\Sigma = V \to \mathbb{Z}$ is the set of stores on the variables in V. The concrete domain is thus $\mathbb{S} = 2^{\Sigma}$, ordered by inclusion.

Definition 2.25 (Abstraction of regular commands)**.** Given an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$, the abstract semantics of regular expressions $[\![\cdot]\!]^{\sharp}_{A} : \mathsf{Reg} \to A \to A$ is inductively defined by

$$\begin{aligned} [\![\mathsf{e}]\!]^{\sharp}_{A} a &= [\![\mathsf{e}]\!]^{A} a \\ [\![\mathsf{r}_1; \mathsf{r}_2]\!]^{\sharp}_{A} a &= [\![\mathsf{r}_2]\!]^{\sharp}_{A}([\![\mathsf{r}_1]\!]^{\sharp}_{A} a) \\ [\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^{\sharp}_{A} a &= [\![\mathsf{r}_1]\!]^{\sharp}_{A} a \vee_A [\![\mathsf{r}_2]\!]^{\sharp}_{A} a \\ [\![\mathsf{r}_1^*]\!]^{\sharp}_{A} a &= \bigvee\nolimits_A \left\{([\![\mathsf{r}_1]\!]^{\sharp}_{A})^k a \mid k \in \mathbb{N}\right\} \end{aligned}$$

To perform computation we must define BCAs (Def. 2.17) on of basic expressions, i.e. the abstract counterpart of $[\![\mathsf{e}]\!] : \mathsf{C}^n \to \mathsf{C}^n$, as $[\![\mathsf{e}]\!]^{\sharp}_{A} : \mathsf{A}^n \to \mathsf{A}^n$.

2.2 Local Completeness

Issues to global completeness As mentioned, the issue dual to soundness is completeness: the soundness of an abstract analysis guarantees that all true alarms are caught, but the lack of completeness results in false alarms being reported too. When false alarms overwhelm true ones, the analysis may become poorly reliable. In particular in [21] global completeness for sound analysis of generic software has been shown to be hard to realise in the setting of Abstract Interpretation. In the paper the authors prove that completeness holds for all programs in a Turing complete programming language only for trivial abstract domains, namely the identity abstraction (so the abstract semantics coincides with the concrete semantics) and the top abstractions, that makes all programs equivalent. Moreover the authors observed that the skip is always trivially complete and composition, conditional and loop statements preserve the completeness of subprograms, hence the only sources of incompleteness are *assignments* and *Boolean guards*.

The idea is then to introduce a local approach to completeness, focusing on a single execution trace produced by the applications of abstract transfer functions on some input of interest: local completeness is defined by modifying Definition 2.18:

Definition 2.26 (Local Completeness)**.** [32, Def. 4.1] An abstract domain $A \in \mathrm{Abs}(C)$ is **locally complete** for a function $f : C \to C$ on a value $c \in C$, denoted by $\mathbb{C}_c^A(f)$, if $(A \circ f)(c) = (A \circ f \circ A)(c)$.

Incorrectness Logic Understanding whether a reported alarm corresponds to a true alarm, which means indirectly understanding whether the approximation is complete, reduces to the challenge of proving some sort of program incorrectness. In [28] the author introduces a simple logic for program incorrectness which is, in a sense, the other side of the coin to Hoare's logic of correctness [1]. These logics are all based on the concept of Hoare triples, which are triples of the form $[p]\mathsf{r}[q]$, in which p, q are assertions and r is a command. The idea of Hoare logic is that provable triples are those in which if the assertion p (that is called the precondition) holds before executing r, then assertion q (the postcondition) holds after the execution of r. In the Local Completeness Logic this idea is relaxed by including the abstraction.

Local Completeness Logic The main theme discussed in [32] is to combine under- and over- approximations and a novel notion of local completeness to define a program logic whose triples either prove correctness or incorrectness. In particular a novel inference rule is introduced to derive all and only those local completeness proofs that are relative to a specific computation trace.

2.2.1 Local Completeness Logic

In [32] the authors define a proof system for program analysis of regular commands that aims at combining over- and under- approximation: this combination is concretised in the logic by the (relax) rule. This proof system is parametrised by an abstraction A, whose provable triples $\vdash_A [p]\ \mathsf{r}\ [q]$ guarantee that

1. q is an under-approximation of $[\![\mathsf{r}]\!]p$, i.e. $q \leq [\![\mathsf{r}]\!]p$;

2. $[\![\mathsf{r}]\!]$ is locally complete for input p and abstraction A, i.e $\mathbb{C}_p^A(\mathsf{r})$ holds;

3. q and $[\![r]\!]p$ have the same over-approximation in A, i.e. $A(q) = A([\![r]\!]p)$.

LCL$_A$ rules. The logical proof system, called **Local Completeness Logic on** A, for short LCL$_A$, includes the rules in Table 2.2.1.

Following [32] we briefly explain the key rules in LCL$_A$, namely (transfer) and (relax), and then explain how these rules let us prove both correctness and incorrectness if there is local completeness.

The rule (transfer) checks that a basic expression e (in the case of the while language a Boolean test or an assignment) is locally complete on p before inferring the result of computing $[\![\mathsf{e}]\!]$ on p as post-condition.

$$\frac{\mathbb{C}^A_p(\mathsf{e})}{\vdash_A [p]\ \mathsf{e}\ [[\![\mathsf{e}]\!]p]}\ \text{(transfer)} \qquad \frac{p' \leq p \leq A(p') \quad \vdash_A [p']\ \mathsf{r}\ [q'] \quad q \leq q' \leq A(q)}{\vdash_A [p']\ \mathsf{r}\ [q']}\ \text{(relax)}$$

$$\frac{\vdash_A [p]\ \mathsf{r}_1\ [w] \quad \vdash_A [w]\ \mathsf{r}_2\ [q]}{\vdash_A [p]\ \mathsf{r}_1; \mathsf{r}_2\ [q]}\ \text{(seq)} \qquad \frac{\vdash_A [p]\ \mathsf{r}_1\ [q_1] \quad \vdash_A [p]\ \mathsf{r}_2\ [q_2]}{\vdash_A [p]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [q_1 \vee q_2]}\ \text{(join)}$$

$$\frac{\vdash_A [p]\ \mathsf{r}\ [w] \quad \vdash_A [p \vee w]\ \mathsf{r}^*\ [q]}{\vdash_A [p]\ \mathsf{r}^*\ [q]}\ \text{(rec)} \qquad \frac{\vdash_A [p]\ \mathsf{r}\ [q] \quad q \leq A(p)}{\vdash_A [p]\ \mathsf{r}^*\ [p \vee q]}\ \text{(iterate)}$$

Table 2.1: Basic rules of the Local Completeness Logic.

The consequence rule (relax) is the key rule of LCL$_A$ as it allows us to combine over- and under-approximating reasoning: it allows us to infer a post-condition that defines an under-approximation q of the exact behaviour and a sound over-approximation $A(q)$ of that exact behaviour. Like in the consequence rules of incorrectness logic by O'Hearn [28] the logical ordering between pre-conditions $p' \Rightarrow p$ and post-conditions $q \Rightarrow q'$ in the premises of (relax) is reversed with respect to the canonical consequence rule of classical Hoare logic, and this is necessary because the post-conditions q in this logic are always under-approximations. The key distinction introduced with the (relax) rule is to constrain the under-approximating post-condition q to also have the same abstraction as the strongest post-conditions, and this lets us preserve the local completeness, hence guaranteeing that any triple derivable in LCL$_A$ either proves correctness or incorrectness. The validity of the rule (relax) relies on observing that local completeness is a kind of "abstract convex property," which means that, if A is locally complete for some $c \in C$, then A is locally complete for all $d \in C$ such that $c \leq d \leq A(c)$ holds.

Proving correctness and incorrectness. Recall that, given a correctness specification Spec, the Abstract Interpretation raises an alarm when $\gamma([\![\mathsf{r}]\!]^\sharp_A)\alpha(p) \not\subseteq \text{Spec}$. This alarm is false if $[\![\mathsf{r}]\!]p \subseteq \text{Spec}$, and true otherwise. The three properties of the provable triples of LCL$_A$ guarantee that we can distinguish between true and false alarms, since we fall into one of the following three cases:

- Case 1: if $\gamma([\![\mathsf{r}]\!]^\sharp_A)\alpha(p) \subseteq \text{Spec}$ then the abstraction does not raise any alarm and the program does not exhibit unwanted behaviours. This holds for any sound (possibly incomplete) Abstract Interpretation.

- Case 2: if Spec is expressible in A and $\gamma([\![\mathsf{r}]\!]^\sharp_A)\alpha(p) \not\subseteq \text{Spec}$ then by local completeness any provable triple $\vdash_A [p]\ \mathsf{r}\ [q]$ is such that all the states in $q \setminus \text{Spec} \neq \varnothing$ are true alarms. If the Abstract Interpretation were not complete we could not distinguish whether the alarms in $\gamma([\![\mathsf{r}]\!]^\sharp_A)\alpha(p) \setminus \text{Spec}$ are false or not.

- Case 3: if Spec is expressible in A and $\gamma([\![\mathsf{r}]\!]^\sharp_A)\alpha(p) \not\subseteq \text{Spec}$ but some proof obligations of local completeness necessary in the proof derivations are not met, then the abstraction A is not precise

$$\dfrac{\dfrac{\vdots}{\vdash_A [\{0,1,4\}]\ (0 < x?); (x := x - 2)\ [\{-1, 2\}]}\,\text{(seq)} \qquad \dfrac{\vdots}{\vdash_A [\{0,1,4\}]\ (0 \geq x?); (x := -x)[\{0\}]}\,\text{(seq)}}{\vdash_A [\{0,1,4\}]\ ((0 < x?); (x := x - 2)) \oplus ((0 \geq x?); (x := -x))\ [\{-1, 0, 2\}]}\,\text{(join)}$$

$$\dfrac{\dfrac{\mathbb{C}^A_{\{0,1,4\}}(0 < x?)}{\vdash_A [\{0,1,4\}]\ (0 < x?)\ [\{1,4\}]}\,\text{(trsf)} \qquad \dfrac{\mathbb{C}^A_{\{1,4\}}(x := x - 2)}{\vdash_A [\{1,4\}]\ (x := x - 2)\ [\{-1,2\}]}\,\text{(trsf)}}{\vdash_A [\{0,1,4\}]\ (0 < x?); (x := x - 2)\ [\{-1, 2\}]}\,\text{(seq)}$$

$$\dfrac{\dfrac{\mathbb{C}^A_{\{0,1,4\}}(x \geq x?)}{\vdash_A [\{0,1,4\}]\ (0 \geq x?)\ [\{0\}]}\,\text{(trsf)} \qquad \dfrac{\mathbb{C}^A_{\{0\}}(x := -x)}{[\{0\}]\ (x := -x)\ [\{0\}]}\,\text{(trsf)}}{\vdash_A [\{0,1,4\}]\ (0 \geq x?); (x := -x)[\{0\}]}\,\text{(seq)}$$

Figure 2.1: Parts of the LCL_A proof tree for a simple Imp program.

enough to distinguish true and false alarms for r on p. One could exploit the failed proof obligations to refine the abstraction enhancing the precision.

We show in the following example a derivation with the Imp language using the Interval abstraction, which abstracts a set of integers representing it with its convex closure: $\alpha(X) = [\inf(X), \sup(X)]$.

Example 2.27. [32, Ex. 4.2] Consider the Imp program

$$\begin{aligned} \mathsf{r} &= \text{if } (0 < x) \text{ then } x := x - 2 \text{ else } x := -x \\ &= ((0 < x?); (x := x - 2)) \oplus ((0 \geq x?); (x := -x)) \end{aligned}$$

and the interval abstraction. The transfer function $[\![0 < x?]\!]$ is not globally complete, but it is on sets p that satisfy one of the following conditions: $p \subseteq \mathbb{Z}_{>0}$, $p \subseteq \mathbb{Z}_{\leq 0}$, $\{0, 1\} \subseteq p$. In Figure 2.1 show the derivation tree for the computation of r on input $\{0, 1, 4\}$, for which the third condition holds.

Part II

Model Checking as Program Analysis

Chapter 3

Model Checking as Program Analysis

The idea we explore in this thesis work is to view Model Checking as an instance of program verification in order to allow for the reuse of the vast theory and toolset of Abstract Interpretation. Given a specification logic, the idea is to map properties expressed in the logic into programs written in a suitable programming language, whose semantics consists of counterexamples to the validity of the formula. We focus on ACTL, the universal fragment of the Computation Tree Logic, in which path properties are universally quantified.

We show that ACTL formulae can be encoded as regular commands in such a way that there is a counterexample to a formula if and only if the semantics of the associated regular commands is $\perp$. In order to do so, we identify a language, called mocha (for Model Checking as Abstract Interpretation).

3.1 ACTL Counterexamples

Since we are interested in a *Model Checking* problem, i.e. in finding *counterexamples* to ACTL specifications, we explicitly characterise counterexamples to the validity of ACTL formulae:

$$
\begin{array}{rcl}
\sigma \not\models \mathsf{p} & \text{iff} & \sigma \vdash \neg\mathsf{p} \\
\sigma \not\models \neg\mathsf{p} & \text{iff} & \sigma \vdash \mathsf{p} \\
\sigma \not\models \varphi_1 \wedge \varphi_2 & \text{iff} & \sigma \not\models \varphi_1 \text{ or } \sigma \not\models \varphi_2 \\
\sigma \not\models \varphi_1 \vee \varphi_2 & \text{iff} & \sigma \not\models \varphi_1 \text{ and } \sigma \not\models \varphi_2 \\
\sigma \not\models \mathsf{AX}\ \varphi_1 & \text{iff} & \exists \sigma \rightarrowtriangle \sigma'.\sigma' \not\models \varphi_1 \\
\sigma \not\models \mathsf{AF}\ \varphi_1 & \text{iff} & \exists \sigma = \sigma_0 \rightarrowtriangle \cdots \rightarrowtriangle \sigma_n \rightarrowtriangle \cdots .\forall k \in \mathbb{N}.\sigma_k \not\models \varphi_1 \\
\sigma \not\models \mathsf{AG}\ \varphi_1 & \text{iff} & \exists \sigma \rightarrowtriangle^* \sigma'.\sigma' \not\models \varphi_1 \\
\sigma \not\models \varphi_1\ \mathsf{AU}\ \varphi_2 & \text{iff} & \exists \sigma = \sigma_0 \rightarrowtriangle \cdots \rightarrowtriangle \sigma_n \rightarrowtriangle \cdots . \\
 & & \forall k \in \mathbb{N}.(\sigma_k \not\models \varphi_2 \text{ or } \exists j \in [0, k-1].\sigma_j \not\models \varphi_1) \\
 & \text{iff} & \exists \sigma = \sigma_0 \rightarrowtriangle \cdots \rightarrowtriangle \sigma_n \rightarrowtriangle \cdots . \\
 & & (\forall k \in \mathbb{N}.\sigma_k \not\models \varphi_2) \text{ or } (\exists j \in \mathbb{N}.\sigma_j \not\models \varphi_1 \text{ and } (\forall i \in [0, j].\sigma_i \not\models \varphi_2))
\end{array}
$$

Remark 3.1. By Remark 1.9 we can further elaborate on the characterization to express them in terms of finite paths:

$$
\begin{array}{rll}
\sigma \not\models \mathsf{AF}\ \varphi_1 & \text{iff} & \exists \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n \twoheadrightarrow \cdots .\forall k \in \mathbb{N}.\sigma_k \not\models \varphi_1 \\
& \text{iff} & \exists \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n. \\
& & (\forall k \in [0,n].\sigma_k \not\models \varphi_1) \text{ and } (\exists i \in [0,n].\sigma_n \twoheadrightarrow \sigma_i) \\
\sigma \not\models \varphi_1\ \mathsf{AU}\ \varphi_2 & \text{iff} & \exists \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n \twoheadrightarrow \cdots . \\
& & (\forall k \in \mathbb{N}.\sigma_k \not\models \varphi_2) \text{ or } (\exists j \in \mathbb{N}.\sigma_j \not\models \varphi_1 \text{ and } (\forall i \in [0,j].\sigma_i \not\models \varphi_2)) \\
& \text{iff} & \exists \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n. \\
& & ((\forall k \in [0,n].\sigma_k \not\models \varphi_2) \text{ and } (\exists i \in [0,n].\sigma_n \twoheadrightarrow \sigma_i)) \\
& & \ \text{or } ((\exists j \in [0,n].\sigma_j \not\models \varphi_1) \text{ and } (\forall i \in [0,j].\sigma_i \not\models \varphi_2)) \\
& \text{iff} & \exists \sigma = \sigma_0 \twoheadrightarrow \cdots \twoheadrightarrow \sigma_n. \\
& & (\forall k \in [0,n].\sigma_k \not\models \varphi_2) \text{ and } (\sigma_n \not\models \varphi_1 \text{ or } \exists i \in [0,n].\sigma_n \twoheadrightarrow \sigma_i)
\end{array}
$$

3.2 Concrete Domain

Given a transition system $(\Sigma, \mathrm{I}, \twoheadrightarrow)$ we construct the concrete domain for the language by means of stacks that keep track of the current state and the set of traversed states. We use this construction for two reasons: we need to record paths because some operators need to be checked on full paths, and we use stacks because operators are nested, and we want to start a new computation when a nested operator is encountered, and we want to retrieve the previous computation once the nested one is concluded.

Definition 3.2 (Abstract path, stack)**.** An **(abstract) path** is a pair $\langle \sigma, T\rangle \in \Sigma \times 2^{\Sigma}$. We denote by P_Σ the set of abstract paths. A **stack** is a finite sequence of paths, i.e. an element of

$$\mathrm{Stacks}_\Sigma = (\mathrm{P}_\Sigma)^* = \bigcup_n (\mathrm{P}_\Sigma)^n$$

An abstract path $\langle \sigma, T\rangle$ is intended to represent abstractly a computation where $T \subseteq \Sigma$ is the set of traversed states and σ is the current state. Note that the order of the traversed states and possible repetitions are abstracted away as they are irrelevant when checking the satisfaction of a formula.

A stack is the representation of a finite set of nested computations. A stack $S^{(n+1)} \in (\mathrm{P}_\Sigma)^{n+1}$ is thus of the shape

$$S^{n+1} = \langle \sigma, T\rangle :: S^n$$

with $S^n \in (\mathrm{P}_\Sigma)^n$ and we say that n is the *length* of the stack S^n.

Since we consider a collecting semantics, the concrete domain will be the powerset of the class of stacks. More precisely, since a formula is always evaluated on stacks of the same length we consider only subsets of stacks with uniform length.

Definition 3.3 (Concrete domain, concrete element)**.** The **concrete domain** is

$$\mathsf{C} = \bigcup_n 2^{(\mathrm{P}_\Sigma)^n}$$

We will denote with C^n the set of stacks of fixed length n, i.e. $\mathsf{C}^n = 2^{(\mathrm{P}_\Sigma)^n}$.

A **concrete element** is a finite set of stacks of the same length

$$c_n \in \mathsf{C}^n \qquad\qquad c_n = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\}$$

where I is a finite set of indices.

Remark 3.4. The bottom element of C (of each C^n) is $\varnothing$.

3.3 The mocha language

We define a language which is used for producing counterexamples to ACTL formulae. The language is based on Kozen's Kleene algebra with test (Sec. 2.1.3), which consists of the usual regular commands of sequential composition, join (choice command, which we indicate with $\sqcup$), and Kleene iteration. The class of basic expression contains operation for manipulating (extending, filtering) abstract paths and for constructing and deconstructing stacks.

Definition 3.5 (mocha syntax)**.** The syntax of the mocha language can be recursively defined as follows:

$$\mathsf{r} ::= \mathsf{e} \mid \mathsf{r}_1; \mathsf{r}_2 \mid \mathsf{r}_1 \sqcup \mathsf{r}_2 \mid \mathsf{r}_1^*$$

$$\mathsf{e} ::= \mathsf{p}? \mid \neg\mathsf{p}? \mid \mathsf{loop}? \mid \mathsf{next} \mid \mathsf{post} \mid \mathsf{push} \mid \mathsf{pop}$$

Definition 3.6 (mocha basic expressions semantics)**.** Let $c = \langle \sigma, T \rangle :: S \in \mathsf{C}$. We define the semantics for basic expressions of the mocha language as follows:

$$\begin{aligned}
(\!|\mathsf{p}?|\!)\, \langle \sigma, T \rangle :: S &= \{\langle \sigma, T \rangle :: S \mid \sigma \models \mathsf{p}\} \\
(\!|\mathsf{loop}?|\!)\, \langle \sigma, T \rangle :: S &= \{\langle \sigma, T \rangle :: S \mid \sigma \in T\} \\
(\!|\mathsf{next}|\!)\, \langle \sigma, T \rangle :: S &= \{\langle \sigma', \varnothing \rangle :: S \mid \sigma \to \sigma'\} \\
(\!|\mathsf{post}|\!)\, \langle \sigma, T \rangle :: S &= \{\langle \sigma', T \cup \{\sigma\} \rangle :: S \mid \sigma \to \sigma'\} \\
(\!|\mathsf{push}|\!)\, \langle \sigma, T \rangle :: S &= \{\langle \sigma, \varnothing \rangle :: \langle \sigma, T \rangle :: S\} \\
(\!|\mathsf{pop}|\!)\, \langle \sigma, T \rangle :: S &= \{S\}
\end{aligned}$$

We briefly comment on the behaviour of these basics expressions: the expression p? checks the validity of the proposition p, the expression loop? checks if the current state loops back to one of the states in the current trace, the expressions next and post extend the trace by one step in all possible ways, but next discards the trace T and does not keep track of the extensions, while post does. The expression push extends the current stack to start a new analysis, while the expression pop restores the previous trace from the stack.

Then we may define the concrete (collecting) semantics of expressions on sets of stacks of any fixed length:

$$\begin{aligned}
[\![\mathsf{r}]\!] : \mathsf{C}^n \to \mathsf{C}^n &\qquad [\![\mathsf{p}?]\!], [\![\mathsf{loop}?]\!], [\![\mathsf{next}]\!], [\![\mathsf{post}]\!] : \mathsf{C}^n \to \mathsf{C}^n \\
[\![\mathsf{push}]\!] : \mathsf{C}^n \to \mathsf{C}^{(n+1)} &\qquad [\![\mathsf{pop}]\!] : \mathsf{C}^n \to \mathsf{C}^{(n-1)}
\end{aligned}$$

We omit any indication of the length of the stacks on which the functions are acting because the definition does not depend on it, so let $c = \{\langle \sigma_i, T_i \rangle :: S_i \mid i \in I\}$ be a set of stacks of the same length. Then the collecting semantics for basic expressions is the standard additive lifting of the semantics defined on a single stack:

$$\begin{array}{rcl}
[\![\mathsf{p?}]\!]\,\{\langle\sigma_i,T_i\rangle :: S_i \mid i\in I\} & = & \{\langle\sigma_i,T_i\rangle :: S_i \mid \sigma_i \models \mathsf{p},\ i\in I\}\\
[\![\mathsf{loop?}]\!]\,\{\langle\sigma_i,T_i\rangle :: S_i \mid i\in I\} & = & \{\langle\sigma_i,T_i\rangle :: S_i \mid \sigma_i \in T_i,\ i\in I\}\\
[\![\mathsf{next}]\!]\,\{\langle\sigma_i,T_i\rangle :: S_i \mid i\in I\} & = & \{\langle\sigma_i',\varnothing\rangle :: S_i \mid \sigma_i \twoheadrightarrow \sigma_i',\ i\in I\}\\
[\![\mathsf{post}]\!]\,\{\langle\sigma_i,T_i\rangle :: S_i \mid i\in I\} & = & \{\langle\sigma_i',T_i\cup\{\sigma_i\}\rangle :: S_i \mid \sigma_i \twoheadrightarrow \sigma_i',\ i\in I\}\\
[\![\mathsf{push}]\!]\,\{\langle\sigma_i,T_i\rangle :: S_i \mid i\in I\} & = & \{\langle\sigma_i,\varnothing\rangle :: \langle\sigma_i,T_i\rangle :: S_i \mid i\in I\}\\
[\![\mathsf{pop}]\!]\,\{\langle\sigma_i,T_i\rangle :: S_i \mid i\in I\} & = & \{S_i \mid i\in I\}
\end{array}$$

where I is a finite set of natural indices. Observe that, in particular, for all commands we have $[\![e]\!]\varnothing = \varnothing$.

Definition 3.7 (Regular expression semantics, see Def. 2.20)**.** The semantics of regular expressions is defined as follows:

$$\begin{array}{rcl}
[\![\mathsf{r}_1;\mathsf{r}_2]\!]c & = & [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]c)\\
[\![\mathsf{r}_1 \sqcup \mathsf{r}_2]\!]c & = & [\![\mathsf{r}_1]\!]c \vee [\![\mathsf{r}_2]\!]c\\
[\![\mathsf{r}_1^*]\!]c & = & \bigvee\{[\![\mathsf{r}_1]\!]^k c \mid k\in\mathbb{N}\}
\end{array}$$

3.4 ACTL formulae as regular commands

ACTL formulae can be mapped in this language in a way that for a given formula φ and system $(\Sigma, \twoheadrightarrow)$, the semantics of the program associated to φ over the system consists of the counterexamples to the validity of the formula (states $\sigma\in\Sigma$ which do not satisfy the formula φ).

Definition 3.8 (ACTL counterexamples programs)**.** To each formula φ in ACTL we assign a program $\lfloor\overline{\varphi}\rfloor$ that computes counterexamples to $\mathcal{P}$, inductively defined as follows:

$$\begin{array}{rcl}
\lfloor\overline{\mathsf{p}}\rfloor & = & \neg\mathsf{p?}\\
\lfloor\overline{\varphi_1\wedge\varphi_2}\rfloor & = & \lfloor\overline{\varphi_1}\rfloor \sqcup \lfloor\overline{\varphi_2}\rfloor\\
\lfloor\overline{\varphi_1\vee\varphi_2}\rfloor & = & \lfloor\overline{\varphi_1}\rfloor ; \lfloor\overline{\varphi_2}\rfloor\\
\lfloor\overline{\mathsf{AX}\ \varphi_1}\rfloor & = & \mathsf{push};\mathsf{next};\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}\\
\lfloor\overline{\mathsf{AF}\ \varphi_1}\rfloor & = & \lfloor\overline{\varphi_1}\rfloor;\mathsf{push};(\mathsf{post};\lfloor\overline{\varphi_1}\rfloor)^*;\mathsf{loop?};\mathsf{pop}\\
\lfloor\overline{\mathsf{AG}\ \varphi_1}\rfloor & = & \mathsf{push};\mathsf{next}^*;\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}\\
\lfloor\overline{\varphi_1\ \mathsf{AU}\ \varphi_2}\rfloor & = & \lfloor\overline{\varphi_2}\rfloor;\mathsf{push};(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^*;(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}
\end{array}$$

We briefly comment the above clauses (but just the non obvious cases) assuming that each command is executed in the state $\langle\sigma,T\rangle :: S$. As ACTL formulae refer to traces, one can imagine that $\langle\sigma,T\rangle :: S$ represents all traces starting at σ.

- a counterexample to $\mathsf{AX}\ \varphi_1$ is when a counterexample to φ_1 at one of the states reachable from σ (as computed by $\mathsf{next};\lfloor\overline{\varphi_1}\rfloor$). Here we can safely use next instead of post since there is no need of checking for loops is done (and the verification is nested inside push and pop). In particular push and pop are used to guarantee that if a counterexample is found then the output contains the state that violates $\mathsf{X}\varphi_1$;

- a counterexample to $\mathsf{AF}\ \varphi_1$ is when φ_1 does not hold in the current state ($\lfloor\overline{\varphi_1}\rfloor$) and after and there is a maximal trace (checked with the $\mathsf{loop}?$) that traverses only reachable states that do not satisfy φ_1 (they are collected using the command $(\mathsf{post};\lfloor\overline{\varphi_1}\rfloor)^*$);
- a counterexample to $\mathsf{AG}\ \varphi_1$ is when we can reach (via repeatedly applying next) a state which is a counterexample to $\lfloor\overline{\varphi_1}\rfloor$. Again we can forget about the trace since there is no $\mathsf{loop}?$ check);
- a counterexample to $\varphi_1\ \mathsf{AU}\ \varphi_2$ is when φ_2 does not hold in the current state ($\lfloor\overline{\varphi_2}\rfloor$) and by computing traces where the next reachable state does not satisfy φ_2 (via the command $(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^*$) we can find a maximal trace (φ_1 does not hold or $\mathsf{loop}?$);

It is immediate to check that, as expected, $\lfloor\overline{\mathsf{tt}\ \mathsf{AU}\ \varphi}\rfloor = \lfloor\overline{\mathsf{AF}\ \varphi}\rfloor$ as in fact the formulae $\mathsf{tt}\ \mathsf{AU}\ \varphi$ and $\mathsf{AF}\ \varphi$ are equivalent. This will be used in the following proposition.

Proposition 3.9 (Formula satisfaction as program verification)**.** We have that for all $\sigma \in \Sigma$, $T \in 2^\Sigma$, $S \in (\Sigma \times 2^\Sigma)^*$

$$\sigma \models \varphi \qquad \Leftrightarrow \qquad [\![\lfloor\overline{\varphi}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} = \varnothing$$

Proof. We prove that:

$$[\![\lfloor\overline{\varphi}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} = \{\langle\sigma,T\rangle :: S \mid \sigma \not\models \varphi\}$$

We proceed by a routine structural induction on φ. The case $\mathsf{AF}\ \varphi$ is not discussed as AF is a derived operator.

($\varphi = \mathsf{p}$ or $\varphi = \neg\mathsf{p}$). Trivial by definition.

($\varphi = \varphi_1 \wedge \varphi_2$ or $\varphi = \varphi_1 \vee \varphi_2$). Just use De Morgan.

($\varphi = \mathsf{AX}\ \varphi_1$). We assume the inductive hypothesis:

$$\forall\sigma,T,S.[\![\lfloor\overline{\varphi_1}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} = \{\langle\sigma,T\rangle :: S \mid \sigma \not\models \varphi_1\}$$

$$\begin{aligned}
[\![\lfloor\overline{\mathsf{AX}\ \varphi_1}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} &= [\![\mathsf{push};\mathsf{next};\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\,\{\langle\sigma,T\rangle :: S\}\\
&= [\![\mathsf{next};\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\,\{\langle\sigma,\varnothing\rangle :: \langle\sigma,T\rangle :: S\}\\
&= [\![\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\,\{\langle\sigma',\varnothing\rangle :: \langle\sigma,T\rangle :: S \mid \sigma \twoheadrightarrow \sigma'\}\\
&= [\![\mathsf{pop}]\!]\,\{\langle\sigma',\varnothing\rangle :: \langle\sigma,T\rangle :: S \mid \sigma \twoheadrightarrow \sigma',\ \sigma' \not\models \varphi_1\}\\
&= \{\langle\sigma,T\rangle :: S \mid \exists\sigma \twoheadrightarrow \sigma',\ \sigma' \not\models \varphi_1\}\\
&= \{\langle\sigma,T\rangle :: S \mid \sigma \not\models \mathsf{AX}\ \varphi_1\}
\end{aligned}$$

($\varphi = \mathsf{AG}\ \varphi_1$). We assume the inductive hypothesis:

$$\forall\sigma,T,S.[\![\lfloor\overline{\varphi_1}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} = \{\langle\sigma,T\rangle :: S \mid \sigma \not\models \varphi_1\}$$

$$
\begin{aligned}
[\![\lfloor\overline{\mathsf{AG}\ \varphi_1}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} &= [\![\mathsf{push};\mathsf{next}^*;\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\,\{\langle\sigma,T\rangle :: S\}\\
&= [\![\mathsf{next}^*;\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\,\{\langle\sigma,\varnothing\rangle :: \langle\sigma,T\rangle :: S\}\\
&= [\![\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\,([\![\mathsf{next}^*]\!]\,\{\langle\sigma,\varnothing\rangle :: \langle\sigma,T\rangle :: S\})\\
&= [\![\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\left(\bigvee\{[\![\mathsf{next}]\!]^n\,\{\langle\sigma,\varnothing\rangle :: \langle\sigma,T\rangle :: S\}\mid n\in\mathbb{N}\}\right)\\
&= [\![\lfloor\overline{\varphi_1}\rfloor;\mathsf{pop}]\!]\left(\bigcup_{n\in\mathbb{N}}\{\langle\sigma_n,\varnothing\rangle :: \langle\sigma,T\rangle :: S\mid\sigma=\sigma_0\twoheadrightarrow\ldots\twoheadrightarrow\sigma_n\}\right)\\
&= [\![\mathsf{pop}]\!]\left(\bigcup_{n\in\mathbb{N}}\{\langle\sigma_n,\varnothing\rangle :: \langle\sigma,T\rangle :: S\mid\sigma=\sigma_0\twoheadrightarrow\ldots\twoheadrightarrow\sigma_n,\ \sigma_n\not\models\varphi_1\}\right)\\
&= \{\langle\sigma,T\rangle :: S\mid\exists\sigma=\sigma_0\twoheadrightarrow\ldots\twoheadrightarrow\sigma_n,\ \sigma_n\not\models\varphi_1\}\\
&= \{\langle\sigma,T\rangle :: S\mid\sigma\not\models\mathsf{AG}\ \varphi_1\}
\end{aligned}
$$

($\varphi = \varphi_1$ AU φ_2). We assume the inductive hypothesis (for $i\in\{1,2\}$):

$$\forall\sigma,T,S.[\![\lfloor\overline{\varphi_1}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\} = \{\langle\sigma,T\rangle :: S\mid\sigma\not\models\varphi_i\}$$

$$
\begin{aligned}
&[\![\lfloor\overline{\varphi_1\ \mathsf{AU}\ \varphi_2}\rfloor]\!]\,\{\langle\sigma,T\rangle :: S\}\\
&\quad= [\![\lfloor\overline{\varphi_2}\rfloor;\mathsf{push};(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^*;(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}]\!]\,\{\langle\sigma,T\rangle :: S\}\\
&\quad= [\![\mathsf{push};(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^*;(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}]\!]\,\{\langle\sigma,T\rangle :: S\mid\sigma\not\models\varphi_2\}\\
&\quad= [\![(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^*;(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}]\!]\,\{\langle\sigma,\varnothing\rangle\ \langle\sigma,T\rangle :: S\mid\sigma\not\models\varphi_2\}\\
&\quad= [\![(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}]\!]\,([\![(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^*]\!]\,\{\langle\sigma,\varnothing\rangle\ \langle\sigma,T\rangle :: S\mid\sigma\not\models\varphi_2\})\\
&\quad= [\![(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}]\!]\left(\bigvee\{[\![(\mathsf{post};\lfloor\overline{\varphi_2}\rfloor)^n]\!]\,\{\langle\sigma,\varnothing\rangle\ \langle\sigma,T\rangle :: S\mid\sigma\not\models\varphi_2\}\mid n\in\mathbb{N}\}\right)\\
&\quad= [\![(\lfloor\overline{\varphi_1}\rfloor\sqcup\mathsf{loop?});\mathsf{pop}]\!]\left(\bigcup_{n\in\mathbb{N}}\{\langle\sigma_n,\{\sigma_0,\ldots,\sigma_{n-1}\}\rangle :: \langle\sigma,T\rangle :: S\mid\sigma=\sigma_0\twoheadrightarrow\ldots\twoheadrightarrow\sigma_n,\ \forall i\in[0,n]\ \sigma_i\not\models\varphi_2\}\right)\\
&\quad= [\![\mathsf{pop}]\!]\Bigg(\bigcup_{n\in\mathbb{N}}\{\langle\sigma_n,\{\sigma_0,\ldots,\sigma_{n-1}\}\rangle :: \langle\sigma,T\rangle :: S\mid\sigma=\sigma_0\twoheadrightarrow\ldots\twoheadrightarrow\sigma_n,\ \forall i\in[0,n]\ \sigma_i\not\models\varphi_2,\\
&\qquad\qquad(\sigma_n\not\models\varphi_1\ \vee\ \exists i\in[0,n-1]\ \sigma_n=\sigma_i)\}\Bigg)\\
&\quad= \{\langle\sigma,T\rangle :: S\mid\exists\sigma=\sigma_0\twoheadrightarrow\ldots\twoheadrightarrow\sigma_n,\ \forall i\in[0,n]\ \sigma_i\not\models\varphi_2,\ (\sigma_n\not\models\varphi_1\vee\exists i\in\{0,\ldots,n-1\}.\sigma_n=\sigma_i)\}\\
&\quad= \{\langle\sigma,T\rangle :: S\mid\sigma\not\models\varphi_1\ \mathsf{AU}\ \varphi_2\}
\end{aligned}
$$

For the last step observe that finiteness of the transition system plays an essential role. In fact, in order to falsify φ_1 AU φ_2 a state σ has to admit a finite trace $\sigma = \sigma_0 \twoheadrightarrow \cdots \sigma_n$ where φ_2 is never satisfied leading to a state which does not satisfy φ_1, or it can admit an infinite trace where φ_2 is never satisfied: since the transition system is finite such an infinite trace must eventually be a cycle (third case). ☐

We discuss a simple example to show how the computation for the verification of properties work in the concrete domain.

Example 3.10. Consider the transition system in Figure 3.1. Assume we are interested in checking whether the property φ = AF c holds in the initial states of the system. To do so we need to compute

$$\begin{tikzcd} {} \arrow[d] & {} \arrow[d] \\ \mathsf{a} \arrow[d] & \mathsf{b} \arrow[l] \\ \mathsf{c} \arrow[loop below] & \end{tikzcd}$$

Figure 3.1: An example to show how mocha computation works.

$[\![\lfloor\overline{\mathsf{AF\ c}}\rfloor]\!]d$, where $d = \{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle\} \in \mathsf{C}^1$ is the set of initial state of the system, i.e. the concrete element on which the computation starts. Using Definition 3.8 we can obtain that the mocha program associated with φ is

$$\begin{aligned}\lfloor\overline{\varphi}\rfloor = \lfloor\overline{\mathsf{AF\ c}}\rfloor &= \lfloor\overline{\mathsf{c}}\rfloor; \mathsf{push}; (\mathsf{post}; \lfloor\overline{\mathsf{c}}\rfloor)^*; \mathsf{loop}?; \mathsf{pop}\\ &= \neg\mathsf{c}?; \mathsf{push}; (\mathsf{post}; \neg\mathsf{c}?)^*; \mathsf{loop}?; \mathsf{pop}\end{aligned}$$

In order to compute its semantics

$$[\![\lfloor\overline{\mathsf{AF\ c}}\rfloor]\!] = [\![\{\mathsf{a},\mathsf{b}\}?; \mathsf{push}; (\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?)^*; \mathsf{loop}?; \mathsf{pop}]\!]$$

we perform a step-by-step computation from the initial state d:

- $[\![\{\mathsf{a},\mathsf{b}\}?]\!]\,\{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle\} = \{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle\}$
- $[\![\mathsf{push}]\!]\,\{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle\} = \{\langle \mathsf{a}, \varnothing\rangle :: \langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle :: \langle \mathsf{b}, \varnothing\rangle\} = \{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\}$
- $[\![(\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?)^*]\!]\,\{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\} = \bigvee_{n\in\mathbb{N}}\{[\![\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?]\!]^n\,\{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\}\}$

$$\begin{aligned}
[\![\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?]\!]^0\,\{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\} &= \{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\}\\
[\![\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?]\!]^1\,\{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\} &= \{\cancel{\langle \mathsf{c}, \{\mathsf{a}\}\rangle :: S_{\mathsf{a}}}, \langle \mathsf{a}, \{\mathsf{b}\}\rangle :: S_{\mathsf{b}}\}\\
[\![\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?]\!]^2\,\{\langle \mathsf{a}, \{\mathsf{b}\}\rangle :: S_{\mathsf{b}}\} &= \{\cancel{\langle \mathsf{c}, \{\mathsf{a},\mathsf{b}\}\rangle :: S_{\mathsf{b}}}\} = \varnothing\\
[\![(\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?)^*]\!]\,\{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}\} &= \{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}, \langle \mathsf{a}, \{\mathsf{b}\}\rangle :: S_{\mathsf{b}}\}
\end{aligned}$$

- $[\![\mathsf{loop}?]\!]\,\{\langle \mathsf{a}, \varnothing\rangle :: S_{\mathsf{a}}, \langle \mathsf{b}, \varnothing\rangle :: S_{\mathsf{b}}, \langle \mathsf{a}, \{\mathsf{b}\}\rangle :: S_{\mathsf{b}}\} = \varnothing$
- $[\![\mathsf{pop}]\!]\varnothing = \varnothing$

As expected the computation yields the empty set: since the formula is true for the transition system no counterexample is found.

We next verify that the properties $\varphi' = \mathsf{AG\ c}$ and $\varphi'' = \mathsf{a\ AU\ c}$ do not hold for the system. We need to compute $[\![\lfloor\overline{\mathsf{AG\ c}}\rfloor]\!]d$ and $[\![\lfloor\overline{\mathsf{a\ AU\ c}}\rfloor]\!]d$, where again $d = \{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle\} \in C^1$. Let us start with φ': by Definition 3.8 we have

$$\begin{aligned}\lfloor\overline{\varphi'}\rfloor = \lfloor\overline{\mathsf{AG\ c}}\rfloor &= \mathsf{push}; \mathsf{next}^*; \lfloor\overline{\mathsf{c}}\rfloor; \mathsf{pop}\\ &= \mathsf{push}; \mathsf{next}^*; \neg\mathsf{c}?; \mathsf{pop}\end{aligned}$$

Hence to compute the semantics

$$[\![\lfloor\overline{\mathsf{AG\ c}}\rfloor]\!] = [\![\mathsf{push}; \mathsf{next}^*; \{\mathsf{a},\mathsf{b}\}?; \mathsf{pop}]\!]$$

we perform a step-by-step computation from d:

- $[\![\mathsf{push}]\!]\,\{\langle\mathsf{a},\varnothing\rangle,\langle\mathsf{b},\varnothing\rangle\} = \{\langle\mathsf{a},\varnothing\rangle :: \langle\mathsf{a},\varnothing\rangle, \langle\mathsf{b},\varnothing\rangle :: \langle\mathsf{b},\varnothing\rangle\} = \{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\}$
- $[\![\mathsf{next}^*]\!]\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\} = \bigvee_{n\in\mathbb{N}}\{[\![\mathsf{next}]\!]^n\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\}\}$

$$[\![\mathsf{next}]\!]^0\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\} = \{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\}$$
$$[\![\mathsf{next}]\!]^1\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\} = \{\langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{b}}\}$$
$$[\![\mathsf{next}]\!]^2\,\{\langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{b}}\} = \{\langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{b}}\}$$
$$[\![\mathsf{next}^*]\!]\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\} = \{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{b}}\}$$

- $[\![\{\mathsf{a},\mathsf{b}\}?]\!]\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{c},\varnothing\rangle :: S_{\mathsf{b}}\} =$
  $= \{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{b}}\}$
- $[\![\mathsf{pop}]\!]\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{b}}\} = \{S_{\mathsf{a}}, S_{\mathsf{b}}\} = \{\langle\mathsf{a},\varnothing\rangle,\langle\mathsf{b},\varnothing\rangle\}$

Hence the computation correctly yields the initial state: since the formula is false for the system, the initial state is a counterexample. Now for the verification of $\varphi'' = \mathsf{a\ AU\ c}$ we can reuse part of the calculation we have done with $\varphi' = \mathsf{AF\ c}$, since by definition

$$\lfloor\overline{\varphi''}\rfloor = \lfloor\overline{\mathsf{a\ \ AU\ \ c}}\rfloor = \lfloor\overline{\mathsf{c}}\rfloor; \mathsf{push}; (\mathsf{post}; \lfloor\overline{\mathsf{c}}\rfloor)^*; (\lfloor\overline{\mathsf{a}}\rfloor \sqcup \mathsf{loop}?); \mathsf{pop}$$

Since we already now that the $[\![\mathsf{loop}?]\!]$ branch yields the empty set, it suffices to compute $[\![\lfloor\overline{\mathsf{a}}\rfloor]\!]$ on $\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{a},\{\mathsf{b}\}\rangle :: S_{\mathsf{b}}\}$:

- $[\![\{\mathsf{b},\mathsf{c}\}?]\!]\,\{\langle\mathsf{a},\varnothing\rangle :: S_{\mathsf{a}}, \langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}, \langle\mathsf{a},\{\mathsf{b}\}\rangle :: S_{\mathsf{b}}\} = \{\langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\}$
- $[\![\mathsf{pop}]\!]\,\{\langle\mathsf{b},\varnothing\rangle :: S_{\mathsf{b}}\} = \{S_{\mathsf{b}}\} = \{\langle\mathsf{b},\varnothing\rangle\}$

hence the computation correctly yields the initial state for which the property does not hold.

As previously mentioned in Section 1.2.2, not all properties that are not satisfied on a Kripke structure admit linear counterexamples, i.e. a *single path* in the structure that witnesses the failure. In particular, it has been shown that is NP-hard to determine whether a linear counterexample exists or not.

We discuss the following example to show that a counterexample, i.e. a state from which a path that does not satisfy the property originates, can be found with the language mocha.

Example 3.11. [15, Ex. 1.2] Consider the transition system of Example 1.14 in Figure 3.2, with $\mathbf{P} = \{\mathsf{tt},\mathsf{a},\neg\mathsf{a},\mathsf{ff}\}$, where $\sigma_0,\sigma_2 \vdash \mathsf{a}$, $\sigma_1 \nvdash \mathsf{a}$. We want to check that the property $\mathsf{AF\ AG\ a}$ does not hold, and to do so we need to compute $[\![\lfloor\overline{\mathsf{AF\ AG\ a}}\rfloor]\!]c_1$, where $c_1 = \{\langle\sigma_0,\varnothing\rangle\} \in \mathsf{C}^1$ is the initial state of the system. By definition we have

$$\lfloor\overline{\mathsf{AF\ AG\ a}}\rfloor = \lfloor\overline{\mathsf{AG\ a}}\rfloor; \mathsf{push}; (\mathsf{post}; \lfloor\overline{\mathsf{AG\ a}}\rfloor)^*; \mathsf{loop}?; \mathsf{pop}$$
$$\lfloor\overline{\mathsf{AG\ a}}\rfloor = \mathsf{push}; \mathsf{next}^*; \lfloor\overline{\mathsf{a}}\rfloor; \mathsf{pop} = \mathsf{push}; \mathsf{next}^*; \neg\mathsf{a}?; \mathsf{pop}$$

$$\begin{tikzcd} {} \arrow[d] & & \\ \sigma_0 \arrow[loop left] \arrow[r] & \sigma_1 \arrow[r] & \sigma_2 \arrow[loop right] \end{tikzcd}$$

Figure 3.2: An example to show how mocha let us compute non-linear counterexamples.

So we need to compute the semantics

$$\begin{aligned} [\![\lfloor\overline{\mathsf{AF\ AG\ a}}\rfloor]\!] &= [\![\lfloor\overline{\mathsf{AG\ a}}\rfloor;\mathsf{push};(\mathsf{post};\lfloor\overline{\mathsf{AG\ a}}\rfloor)^*;\mathsf{loop?};\mathsf{pop}]\!] \\ [\![\lfloor\overline{\mathsf{AG\ a}}\rfloor]\!] &= [\![\mathsf{push};\mathsf{next}^*;\neg\mathsf{a?};\mathsf{pop}]\!] \end{aligned}$$

Performing a step-by-step computation from the initial state c_1, the first thing we need to do is to compute $[\![\lfloor\overline{\mathsf{AG\ a}}\rfloor]\!]c_1$:

- $[\![\mathsf{push}]\!]\,\{\langle\sigma_0,\varnothing\rangle\} = \{\langle\sigma_0,\varnothing\rangle :: \langle\sigma_0,\varnothing\rangle\} = \{\langle\sigma_0,\varnothing\rangle :: S_0\}$
- $[\![\mathsf{next}^*]\!]\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \bigvee_{n\in\mathbb{N}}\{[\![\mathsf{next}]\!]^n\,\{\langle\sigma_0,\varnothing\rangle :: S_0\}\}$

$$\begin{aligned} &[\![\mathsf{next}]\!]^0\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\varnothing\rangle :: S_0\} \\ &[\![\mathsf{next}]\!]^1\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_1,\varnothing\rangle :: S_0\} \\ &[\![\mathsf{next}]\!]^2\,\{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_1,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_1,\varnothing\rangle :: S_0, \langle\sigma_2,\varnothing\rangle :: S_0\} \\ &[\![\mathsf{next}^*]\!]\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_1,\varnothing\rangle :: S_0, \langle\sigma_2,\varnothing\rangle :: S_0\} \end{aligned}$$

- $[\![\neg\mathsf{a?}]\!]\,\{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_1,\varnothing\rangle :: S_0, \langle\sigma_2,\varnothing\rangle :: S_0\} = \{\langle\sigma_1,\varnothing\rangle :: S_0\}$
- $[\![\mathsf{pop}]\!]\,\{\langle\sigma_1,\varnothing\rangle :: S_0\} = \{S_0\} = \{\langle\sigma_0,\varnothing\rangle\}$

Then we have

- $[\![\mathsf{push}]\!]\,\{\langle\sigma_0,\varnothing\rangle\} = \{\langle\sigma_0,\varnothing\rangle :: S_0\}$
- $[\![(\mathsf{post};\lfloor\overline{\mathsf{AG\ a}}\rfloor)^*]\!]\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \bigvee_{n\in\mathbb{N}}\{[\![(\mathsf{post};\lfloor\overline{\mathsf{AG\ a}}\rfloor]\!]^n\,\{\langle\sigma_0,\varnothing\rangle :: S_0\}\}$

$$\begin{aligned} &[\![(\mathsf{post};\lfloor\overline{\mathsf{AG\ a}}\rfloor)]\!]^0\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\varnothing\rangle :: S_0\} \\ &[\![(\mathsf{post};\lfloor\overline{\mathsf{AG\ a}}\rfloor)]\!]^1\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\{\sigma_0\}\rangle :: S_0, \langle\sigma_1,\{\sigma_0\}\rangle :: S_0\} \\ &[\![(\mathsf{post};\lfloor\overline{\mathsf{AG\ a}}\rfloor)^*]\!]\,\{\langle\sigma_0,\varnothing\rangle :: S_0\} = \{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_0,\{\sigma_0\}\rangle :: S_0, \langle\sigma_1,\{\sigma_0\}\rangle :: S_0\} \end{aligned}$$

- $[\![\mathsf{loop?}]\!]\,\{\langle\sigma_0,\varnothing\rangle :: S_0, \langle\sigma_0,\{\sigma_0\}\rangle :: S_0, \langle\sigma_1,\{\sigma_0\}\rangle :: S_0\} = \{\langle\sigma_0,\{\sigma_0\}\rangle :: S_0\}$
- $[\![\mathsf{pop}]\!]\,\{\langle\sigma_0,\{\sigma_0\}\rangle :: S_0\} = \{S_0\} = \{\langle\sigma_0,\{\sigma_0\}\rangle\}$

Chapter 4

Injecting Abstraction

We want to study how to enforce abstractions on this concrete domain starting from abstractions on the states of the transition system. We first present an abstraction that enables us to simplify the computations in the concrete domain, by merging traces relative to the same state and the same past (encoded in the rest of the stack).

We then present two ways of lifting a given abstraction on states to an abstraction on stacks. In the first way, in which we consider an abstraction that induces a partition of states, we build the stack domain using the construction given in Chapter 3 starting from the abstract transition system, i.e. the system in which some states are identified according to the partition.

In the second way we induce a much coarser abstraction by first flattening the set of stacks grouping together all the current states and all the current traces (and recursively on the rest of the stack) an then applying the abstraction (that does not need to be a partition) on the obtained two sets (one for states, one for traces). The result is a coarse abstraction, because we lose all the relations between a state, its trace and its past encoded in the stack.

Recall that, given a transition system $(\Sigma, \mathrm{I}, \twoheadrightarrow)$, we built the concrete domain as

$$\mathsf{C} = \bigcup_{n \in \mathbb{N}} \left(2^{\mathrm{P}_\Sigma}\right)^n = \bigcup_{n \in \mathbb{N}} \mathsf{C}^n$$

and we defined stacks as $S^n = \langle \sigma, T \rangle :: S^{(n-1)}$ for $n \geq 2$ where $S^1 = \langle \sigma', T' \rangle$. Hence concrete elements are set of stacks of fixed length:

$$\mathsf{C}^n \ni c_n = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N} \right\}$$

We are interested in particular in studying the abstract semantics of basic expressions to understand which conditions we should require to achieve completeness (at least for basic expressions). The abstract semantics of basic expressions is defined as the *best correct approximation*, i.e.

$$[\![\mathsf{e}]\!]^\sharp_A = [\![\mathsf{e}]\!]^A = \alpha \circ [\![\mathsf{e}]\!] \circ \gamma$$

having the following picture in mind:

$$\begin{tikzcd} A \arrow[r, "\llbracket \mathsf{e} \rrbracket^\sharp_A"] \arrow[d, bend left, "\gamma"] & A \arrow[d, bend left, "\gamma"] \\ C \arrow[r, "\llbracket \mathsf{e} \rrbracket"] \arrow[u, bend left, "\alpha"] & C \arrow[u, bend left, "\alpha"] \end{tikzcd}$$

In particular completeness holds by definition if $\alpha \circ \llbracket \mathsf{e} \rrbracket = \llbracket \mathsf{e} \rrbracket^\sharp_A \circ \alpha$.

4.1 A local completeness-preserving abstraction

We show that we can simplify the description of a nested computation, preserving local completeness, by means of an abstraction that merges the traces relative to the same state. The only command that makes use of the trace is loop?, which is, in the translation of ACTL into mocha, always paired with the pop command. For this reason in this section we will consider the composite command loop?; pop in the proof of completeness of basic commands. This abstraction is globally complete for basic expression, and we will exploit this fact to us in as an underlying simplification of other abstractions.

The idea that justifies the introduction of this abstraction is that we are interested in keeping the set of traversed states, i.e. the "past" of the current state of the current (nested) computation only for checking the existence of a cycle. We noticed that the check for cycles (loop?) is always followed by a pop command that restores the state the nested computation started from: this led us to consider that a simplification of the domain was possible, by unifying all the stacks that are compatible in the following sense. The identification must ensure that the stacks retrieved by the combined command loop?; pop are all and only those that would actually include a loop: for this reason we need to check not only that the current state is the same, but also the tail of the stack should coincide to perform the unification. This abstraction must then be performed from the deepest level of the stack to the top level (in the reverse order no unification would be possible).

We introduce some auxiliary functions to build in steps the abstraction and the concretization map.

Definition 4.1 (Pop map)**.** Define the **pop map** as pop : $\mathsf{C}^{n+1} \to \mathsf{C}^n$ as

$$\mathrm{pop}(X) = \{S \mid \langle \sigma, T \rangle :: S \in X\}$$

The map $\mathrm{pop}^{n-i}(X)$ eliminates the first $n-i$ elements of the stack.

Definition 4.2 (Merge map)**.** Define the **merge map** as $\mathrm{m}_i : \mathsf{C}^n \to \mathsf{C}^n$ as

$$\mathrm{m}_i(X) = \Big\{ S_1 :: \langle \sigma, T \rangle :: S_2 \mid S_1 \in \mathsf{C}^{n-i},\ S_2 \in \mathsf{C}^{i-1},\ S_1 :: \langle \sigma, T' \rangle :: S_2 \in X, \\ T = \bigcup \{T'' \mid \langle \sigma, T'' \rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\} \Big\}$$

Definition 4.3 (Abstraction map)**.** The **abstraction map** for stacks of length n is defined as

$$\alpha_n : \mathsf{C}^n \to \mathsf{C}^n \qquad \alpha_n = \mathrm{m}_n \circ \cdots \circ \mathrm{m}_1$$

Definition 4.4 (Splitting map)**.** Define the **splitting map** as $\mathrm{s}_i : \mathsf{C}^n \to \mathsf{C}^n$ as

$$\mathrm{s}_i(Y) = \big\{ S_1 :: \langle \sigma, T' \rangle :: S_2 \mid S_1 \in \mathsf{C}^{n-i},\ S_2 \in \mathsf{C}^{i-1},\ S_1 :: \langle \sigma, T \rangle :: S_2 \in Y,\ T \subseteq T' \big\}$$

Definition 4.5. The **concretization map** for stacks of length n is defined as

$$\gamma_n : \mathsf{C}^n \to \mathsf{C}^n \qquad \gamma_n = \mathrm{s}_1 \circ \cdots \circ \mathrm{s}_n$$

In the following we will indicate α_n, γ_n with the indices to make the computations more explicit, but we understand that the functions we are actually interested in are $\alpha : \mathsf{C} \to \mathsf{A}$, $\gamma : \mathsf{A} \to \mathsf{C}$ defined as:

$$\alpha = \bigoplus_{n \in \mathbb{N}} \alpha_n \qquad \gamma = \bigoplus_{n \in \mathbb{N}} \gamma_n$$

Remark 4.6. With a little abuse of notation (since in principle each $\mathrm{m}_i, \mathrm{s}_j$ should depend on two indices, the second being n, the length of the stacks it is operating on) we think of α_n, γ_n as recursively defined as follows:

$$\alpha_1 = \mathrm{m}_1 \qquad \alpha_n = \mathrm{m}_n \circ \alpha_{n-1}$$
$$\gamma_1 = \mathrm{s}_1 \qquad \gamma_n = \mathrm{s}_n \circ \gamma_{n-1}$$

where, letting $b = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\} \in \mathsf{C}^n$, we can explicitly compute

$$\alpha_n(b) = \mathrm{m}_n \circ \alpha_{n-1}(b) = \mathrm{m}_n \left(\{ \langle \sigma_i, T_i \rangle :: S_i' \mid i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(b)) \} \right)$$
$$\gamma_n(b) = \mathrm{s}_n \circ \gamma_{n-1}(b) = \mathrm{s}_n \left(\left\{ \langle \sigma_i, T_i \rangle :: S_i' \mid i \in I,\ S_i' \in \gamma_{n-1} \left(\left\{ S_i^{(n-1)} \right\} \right) \right\} \right)$$

where for the abstraction we need to consider the whole set because we want to unify traces from different stacks, while for the closure we just need to look at subsets relative to a unique stack. We can further compute

$$\mathrm{m}_n(b) = \left\{ \left\langle \sigma_i, \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\rangle :: S_i^{(n-1)} \mid i \in I \right\}$$

which will be useful in the following analysis of the local completeness of the basic expressions.

4.1.1 Galois Connection

In order to prove that $(\alpha_n, \mathsf{C}^n, \mathsf{C}^n, \gamma_n)$ is a Galois insertion we show that $\gamma_n \circ \alpha_n$ is a closure operator, i.e. it is monotone, idempotent and extensive. We show that each $\mathrm{s}_i \circ \mathrm{m}_i$ is a closure operator, and that some commutation relations hold.

Definition 4.7 (Commutation of functions)**.** Let X be a set, $f, g : X \to X$. We say that f, g **commute** if $f \circ g = g \circ f$.

Proposition 4.8. We have the following commutations relations:

1. s_i and s_j commute for all $i, j \in \mathbb{N}$;

2. s_j commutes with m_i for $j > i$;

3. $\mathrm{s}_i \circ \mathrm{m}_i$ and $\mathrm{s}_j \circ \mathrm{m}_j$ commute for any $i, j \in \mathbb{N}$.

Proof. **1.** Without loss of generality, assume $j > i$. Recall that

$$\mathrm{s}_i(Y) = \{S_1 :: \langle \sigma, T' \rangle :: S_2 \mid S_1 \in \mathsf{C}^{n-i},\ S_2 \in \mathsf{C}^{i-1},\ S_1 :: \langle \sigma, T \rangle :: S_2 \in Y,\ T \subseteq T'\}$$

Then

$$\mathrm{s}_j(\mathrm{s}_i(Y)) = \{S_1' :: \langle \rho, V' \rangle :: S_3 :: \langle \sigma, T' \rangle :: S_2 \mid S_1' \in \mathsf{C}^{n-j},\ S_3 \in \mathsf{C}^{j-i-1},\ S_2 \in \mathsf{C}^{i-1},$$
$$S_1' :: \langle \rho, V \rangle :: S_3 :: \langle \sigma, T \rangle :: S_2 \in Y,\ T \subseteq T',\ V \subseteq V'\} = \mathrm{s}_j(\mathrm{s}_i(Y))$$

2. Recall that

$$\mathrm{m}_i(X) = \left\{S_1 :: \langle \sigma, T \rangle :: S_2 \mid S_1 \in \mathsf{C}^{n-i},\ S_2 \in \mathsf{C}^{i-1},\ S_1 :: \langle \sigma, T' \rangle :: S_2 \in X, T = \bigcup \{T'' \mid \langle \sigma, T'' \rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\}\right\}$$

Then

$$\mathrm{s}_j(\mathrm{m}_i(X)) = \left\{S_1' :: \langle \rho, V' \rangle :: S_3 :: \langle \sigma, T \rangle :: S_2 \mid S_1' \in \mathsf{C}^{n-j},\ S_3 \in \mathsf{C}^{j-i-1},\ S_2 \in \mathsf{C}^{i-1},\right.$$
$$\left. S_1' :: \langle \rho, V \rangle :: S_3 :: \langle \sigma, T' \rangle :: S_2 \in X,\ V \subseteq V',\ T = \bigcup \{T'' \mid \langle \sigma, T'' \rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\}\right\}$$
$$= \mathrm{m}_i(\mathrm{s}_j(X))$$

3. Without loss of generality assume $j > i$. Then we may compute $\mathrm{s}_i \circ \mathrm{m}_i$ as

$$\mathrm{s}_i(\mathrm{m}_i(X)) = \left\{S_1 :: \langle \sigma, T' \rangle :: S_2 \mid S_1 \in \mathsf{C}^{n-i},\ S_2 \in \mathsf{C}^{i-1},\right.$$
$$\left. S_1 :: \left\langle \sigma, \hat{T} \right\rangle :: S_2 \in X,\ T' \subseteq T = \bigcup \{T'' \mid \langle \sigma, T'' \rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\}\right\}$$

Then

$$\mathrm{s}_j(\mathrm{m}_j(\mathrm{s}_i(\mathrm{m}_i(X)))) = \left\{S_1' :: \langle \rho, V' \rangle :: S_3 :: \langle \sigma, T' \rangle :: S_2 \mid S_1' \in \mathsf{C}^{n-j},\ S_3 \in \mathsf{C}^{j-i-1},\ S_2 \in \mathsf{C}^{i-1},\right.$$
$$S_1' :: \left\langle \rho, \hat{V} \right\rangle :: S_3 :: \left\langle \sigma, \hat{T} \right\rangle :: S_2 \in X,$$
$$T' \subseteq T = \bigcup \{T'' \mid \langle \sigma, T'' \rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\},$$
$$\left. V' \subseteq V = \bigcup \left\{V'' \mid \langle \rho, V'' \rangle :: S_3 :: \left\langle \sigma, \hat{T} \right\rangle :: S_2 \in \mathrm{pop}^{n-j}(X)\right\}\right\}$$
$$= \mathrm{s}_i(\mathrm{m}_i(\mathrm{s}_j(\mathrm{m}_j(X))))$$

□

These relation are useful because by 2. we have

$$\gamma_n \circ \alpha_n = \mathrm{s}_n \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_n \circ \dots \mathrm{m}_1 = \mathrm{s}_n \circ \mathrm{m}_n \circ g_{n-1} \circ f_{n-1} \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_1$$

while the other properties will be used to prove that $\gamma_n \circ \alpha_n$ is a closure operator.

Proposition 4.9. Each $\mathrm{s}_i \circ \mathrm{m}_i$ is a closure operator.

Proof. **(Monotone)** We need to show that if $X, Y \subseteq \mathsf{C}^n$, $X \subseteq Y$, then $\mathrm{s}_i(\mathrm{m}_i(X)) \subseteq \mathrm{s}_i(\mathrm{m}_i(Y))$. We have that

$$\mathrm{s}_i(\mathrm{m}_i(X)) = \left\{S_1 :: \langle \sigma, T' \rangle :: S_2 \mid S_1 \in \mathsf{C}^{n-i},\ S_2 \in \mathsf{C}^{i-1},\right.$$
$$\left. S_1 :: \left\langle \sigma, \hat{T} \right\rangle :: S_2 \in X,\ T' \subseteq T = \bigcup \{T'' \mid \langle \sigma, T'' \rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\}\right\}$$

Notice first that

$$T_X = \bigcup \{T'' \mid \langle\sigma, T''\rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\} \subseteq \bigcup \{T'' \mid \langle\sigma, T''\rangle :: S_2 \in \mathrm{pop}^{n-i}(Y)\} = T_Y$$

So let $S_1 :: \langle\sigma, T'\rangle :: S_2 \in \mathrm{s}_i(\mathrm{m}_i(X))$, then there exists $\hat{T}$ such that $S_1 :: \left\langle\sigma, \hat{T}\right\rangle :: S_2 \in X$, but then $S_1 :: \left\langle\sigma, \hat{T}\right\rangle :: S_2 \in Y$. Moreover, $T' \subseteq T_X \subseteq T_Y$, hence combining these two facts $S_1 :: \langle\sigma, T'\rangle :: S_2 \in \mathrm{s}_i(\mathrm{m}_i(Y))$.

(Idempotent) In order to show that $(\mathrm{s}_i \circ \mathrm{m}_i)^2 = \mathrm{s}_i \circ \mathrm{m}_i$ we show that $\mathrm{m}_i \circ \mathrm{s}_i \circ \mathrm{m}_i = \mathrm{m}_i$, because then $\mathrm{s}_i \circ (\mathrm{m}_i \circ \mathrm{s}_i \circ \mathrm{m}_i) = \mathrm{s}_i \circ \mathrm{m}_i$, which is what we want. From the previous computations of $\mathrm{s}_i(\mathrm{m}_i(X))$ and the fact that $\bigcup_{T' \subseteq T} T' = T$, we have that $\mathrm{m}_i(\mathrm{s}_i(\mathrm{m}_i(X))) = \mathrm{m}_i(X)$.

(Extensive) We need to show that if $S_1 :: \langle\sigma, T'\rangle :: S_2 \in X$, then $S_1 :: \langle\sigma, T'\rangle :: S_2 \in \mathrm{s}_i(\mathrm{m}_i(X))$. But this follows from the fact that $T' \subseteq T = \bigcup \{T'' \mid \langle\sigma, T''\rangle :: S_2 \in \mathrm{pop}^{n-i}(X)\}$. □

Proposition 4.10. The composite map $\gamma_n \circ \alpha_n$ is a closure operator.

Proof. **(Monotone)** The map $\gamma_n \circ \alpha_n$ is monotone since it is the composition of monotone maps.

(Idempotent) We have

$$\begin{aligned}
\gamma_n \circ \alpha_n \circ \gamma_n \circ \alpha_n &= \mathrm{s}_n \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_n \circ \dots \mathrm{m}_1 \circ \mathrm{s}_n \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_n \circ \dots \mathrm{m}_1 \\
&= \mathrm{s}_n \circ \mathrm{m}_n \circ g_{n-1} \circ f_{n-1} \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_1 \circ \mathrm{s}_n \circ \mathrm{m}_n \circ g_{n-1} \circ f_{n-1} \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_1 \\
&= (\mathrm{s}_n \circ \mathrm{m}_n)^2 \circ (g_{n-1} \circ f_{n-1})^2 \circ \dots \circ (\mathrm{s}_1 \circ \mathrm{m}_1)^2 \\
&= \mathrm{s}_n \circ \mathrm{m}_n \circ g_{n-1} \circ f_{n-1} \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_1 \\
&= \mathrm{s}_n \circ \dots \circ \mathrm{s}_1 \circ \mathrm{m}_n \circ \dots \mathrm{m}_1 = \gamma_n \circ \alpha_n
\end{aligned}$$

since $\mathrm{s}_i \circ \mathrm{m}_i$ is idempotent.

(Extensive) It follows from the fact that each $\mathrm{s}_i \circ \mathrm{m}_i$ is extensive, and by the fact that $\mathrm{s}_i \circ \mathrm{m}_i$ and $\mathrm{s}_j \circ \mathrm{m}_j$ commute for any i, j. □

4.1.2 Abstract Semantics

We now compute the Best Correct Approximations for basic expressions and show that this abstraction is globally complete. As mentioned above, in order to prove completeness we check that the composite command loop?; pop is complete. This is enough because in the translation of ACTL in the language mocha the only clauses that involve the loop? command are $\lfloor \overline{\mathsf{AF}\ \varphi} \rfloor$ and $\lfloor \overline{\psi\ \mathsf{AU}\ \varphi} \rfloor$, and in both cases this check is immediately followed by a pop, and because the only relevant information is if there exists a loop, since then we are only interested in retrieving the previous element of the stack if the answer is positive, or to discard the stack if the answer is negative.

$[\![\mathsf{p?}]\!]^\sharp_A$ computation Consider $a = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$ such that $\alpha_n(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{p?}]\!]^\sharp_A(a) &= (\alpha_n[\![\mathsf{p?}]\!]\gamma_n)(a) = (\alpha_n[\![\mathsf{p?}]\!]\gamma_n)\left(\left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}\right)\\
&= (\alpha_n[\![\mathsf{p?}]\!])\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \alpha_n\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid \sigma_i \models \mathsf{p},\ i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid \sigma_i \models \mathsf{p},\ i \in I\right\}
\end{aligned}$$

where the last equality holds because the union of all subsets of a set X is precisely the set X.

Now to prove completeness we must show that $\alpha_n \circ [\![\mathsf{p?}]\!]_n = [\![\mathsf{p?}]\!]^{\sharp n}_A \circ \alpha_n$.

Consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$, then:

$$\begin{aligned}
(\alpha_n \circ [\![\mathsf{p?}]\!])(c_n) &= \alpha_n\left(\left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid \sigma_i \models \mathsf{p},\ i \in I\right\}\right)\\
&= \mathrm{m}_n\left(\{\langle\sigma_i, T_i\rangle :: S_i' \mid \sigma_i \models \mathsf{p},\ i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c))\}\right)\\
&= \left\{\left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid \sigma_i \models \mathsf{p},\ i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c))\right\}\\
[\![\mathsf{p?}]\!]^\sharp_A \circ \alpha_n(c_n) &= [\![\mathsf{p?}]\!]^\sharp_A\left(\left\{\left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n))\right\}\right)\\
&= \left\{\left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid \sigma_i \models \mathsf{p},\ i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n))\right\}
\end{aligned}$$

Thus we can conclude that the domain is globally complete for $\mathsf{p?}$.

$[\![\mathsf{loop?};\mathsf{pop}]\!]^\sharp_A$ computation Consider $a = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$ such that $\alpha_n(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{loop?};\mathsf{pop}]\!]^\sharp_A(a) &= (\alpha_n[\![\mathsf{loop?};\mathsf{pop}]\!]\gamma_n)(a) = (\alpha_n[\![\mathsf{loop?};\mathsf{pop}]\!]\gamma_n)\left(\left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}\right)\\
&= (\alpha_n[\![\mathsf{loop?};\mathsf{pop}]\!])\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= (\alpha_n[\![\mathsf{pop}]\!])\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ \sigma_i \in T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \alpha_n\left(\{S_i' \mid i \in I,\ \sigma_i \in T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \{S_i \mid i \in I, \sigma_i \in T_i,\ i \in I\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_{n-1} \circ [\![\mathsf{loop?};\mathsf{pop}]\!]_n = [\![\mathsf{loop?};\mathsf{pop}]\!]^{\sharp n}_A \circ \alpha_n$.

Consider $c_n = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha_{n-1} \circ [\![\mathsf{loop?};\mathsf{pop}]\!])(c_n) &= \alpha_{n-1}[\![\mathsf{pop}]\!]\left(\left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I,\ \sigma_i \in T_i\right\}\right)\\
&= \alpha_{n-1}\left(\{S_i \mid i \in I,\ \sigma_i \in T_i\}\right)\\
[\![\mathsf{loop?};\mathsf{pop}]\!]^\sharp_A \circ \alpha_n(c_n) &= [\![\mathsf{loop?};\mathsf{pop}]\!]^\sharp_A \left(\left\{\left\langle \sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n))\right\}\right)\\
&= \alpha_{n-1}\left(\left\{S_i' \mid i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n)),\ \sigma_i \in \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\}\right)\\
&= \alpha_{n-1}\left(\{S_i \mid i \in I,\ \sigma_i \in T_i\}\right)
\end{aligned}$$

where the last equality holds because it is irrelevant to the computation of α_{n-1} if the unification had happened before. Thus we can conclude that the domain is globally complete for $\mathsf{loop?};\mathsf{pop}$.

Remark 4.11. For the following operations $\mathsf{post}, \mathsf{next}$ we study separately the cases $a \in \mathsf{C}^1$ and $a \in \mathsf{C}^n$ to understand the computation more clearly.

$[\![\mathsf{next}]\!]^\sharp_A$ computation, case $a \in \mathsf{C}^1$ Consider $a = \{\langle \sigma_i, T_i\rangle \mid i \in I\}$ such that $\alpha_1(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{next}]\!]^\sharp_A(a) &= (\alpha_1[\![\mathsf{next}]\!]\gamma_1)(a) = (\alpha_1[\![\mathsf{next}]\!]\gamma_1)\left(\{\langle \sigma_i, T_i\rangle \mid i \in I\}\right)\\
&= (\alpha_1[\![\mathsf{next}]\!])\left(\{\langle \sigma_i, T_i'\rangle \mid i \in I,\ T_i' \subseteq T_i,\}\right)\\
&= \alpha_1\left(\{\langle \sigma_i', \varnothing\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i'\}\right)\\
&= \{\langle \sigma_i', \varnothing\rangle \mid \sigma_i \twoheadrightarrow \sigma_i',\ i \in I\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_1 \circ [\![\mathsf{next}]\!]_1 = [\![\mathsf{next}]\!]^{\sharp 1}_A \circ \alpha_1$. Consider $c_1 = \{\langle \sigma_i, T_i\rangle \mid i \in I \subseteq \mathbb{N}\}$, then:

$$\begin{aligned}
(\alpha_1 \circ [\![\mathsf{next}]\!])(c_1) &= \alpha_1\left(\{\langle \sigma_i', \varnothing\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i'\}\right)\\
&= \{\langle \sigma_i', \varnothing\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i'\}\\
[\![\mathsf{next}]\!]^{\sharp 1}_A \circ \alpha_1(c_1) &= [\![\mathsf{next1}]\!]^\sharp_A\left(\left\{\left\langle \sigma_i, \bigcup_{\substack{h\in I\\ \sigma_h=\sigma_i}} T_h\right\rangle \mid i \in I\right\}\right)\\
&= \{\langle \sigma_i', \varnothing\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i'\}
\end{aligned}$$

Thus we can conclude that the domain is globally complete for next.

$[\![\mathsf{next}]\!]^\sharp_A$ computation, case $a \in \mathsf{C}^n$ Consider $a = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$ such that $\alpha_n(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{next}]\!]^\sharp_A(a) &= (\alpha_n[\![\mathsf{next}]\!]\gamma_n)(a) = (\alpha_n[\![\mathsf{next}]\!]\gamma_n)\left(\left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}\right)\\
&= (\alpha_n[\![\mathsf{next}]\!])\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \alpha_n\left(\{\langle\sigma_i', \varnothing\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i', S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \{\langle\sigma_i', \varnothing\rangle :: S_i'' \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i', S_i'' \in \mathrm{pop}\left(\alpha_{n-1}\left(\{\langle\sigma_i', \varnothing\rangle :: S_i \mid i \in I, \sigma_i \twoheadrightarrow \sigma_i'\}\right)\right)\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_n \circ [\![\mathsf{next}]\!]_n = [\![\mathsf{next}]\!]^{\sharp n}_A \circ \alpha_n$. Consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$, then:

$$\begin{aligned}
(\alpha_n \circ [\![\mathsf{next}]\!])(c_n) &= \alpha_n\left(\overbrace{\left\{\langle\sigma_i', \varnothing\rangle :: S_i^{(n-1)} \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i'\right\}}^{c_n'}\right)\\
&= \mathrm{m}_n\left(\{\langle\sigma_i', \varnothing\rangle :: S_i' \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i',\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n'))\}\right)\\
&= \{\langle\sigma_i', \varnothing\rangle :: S_i' \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i',\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n'))\}\\
[\![\mathsf{next}]\!]^\sharp_A \circ \alpha_n(c_n) &= [\![\mathsf{next}]\!]^\sharp_A\left(\left\{\left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n))\right\}\right)\\
&= \{\langle\sigma_i', \varnothing\rangle :: S_i'' \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i',\\
&\qquad S_i'' \in \mathrm{pop}\left(\alpha_{n-1}\left(\{\langle\sigma_i', \varnothing\rangle :: S_i' \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i',\ S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n))\}\right)\right)\}\\
&= \{\langle\sigma_i', \varnothing\rangle :: S_i'' \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i', S_i'' \in \mathrm{pop}(\alpha_{n-1}(c_n'))\}
\end{aligned}$$

where we use the fact that if $\sigma_h = \sigma_j$ and $\sigma_j \twoheadrightarrow \sigma_i'$ then obviously $\sigma_h \twoheadrightarrow \sigma_i'$. The last passage is justified from the fact that ultimately the abstraction is applied to the set of stacks *after* the post has been computed, and it is of no influence the fact that in the second case $S_i' \in \mathrm{pop}(\alpha_{n-1}(c_n))$, since if any unification occurred at that first level, it will happen again in $S_i'' \in \mathrm{pop}(\alpha_{n-1}(c_n'))$ for the same reason that f $\sigma_h = \sigma_j$ and $\sigma_j \twoheadrightarrow \sigma_i'$ then obviously $\sigma_h \twoheadrightarrow \sigma_i'$. Thus we can conclude that the domain is globally complete for next.

$[\![\mathsf{post}]\!]^\sharp_A$ computation, case $a \in \mathsf{C}^1$ Consider $a = \{\langle\sigma_i, T_i\rangle \mid i \in I\}$ such that $\alpha_1(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{post}]\!]^\sharp_A(a) &= (\alpha_1[\![\mathsf{post}]\!]\gamma_1)(a) = (\alpha_1[\![\mathsf{post}]\!]\gamma_1)\left(\{\langle\sigma_i, T_i\rangle \mid i \in I\}\right)\\
&= (\alpha_1[\![\mathsf{post}]\!])\left(\{\langle\sigma_i, T_i'\rangle \mid i \in I,\ T_i' \subseteq T_i, \}\right)\\
&= \alpha_1\left(\{\langle\sigma_i', T_i' \cup \{\sigma_i\}\rangle \mid i \in I,\ T_i' \subseteq T_i,\ \sigma_i \twoheadrightarrow \sigma_i'\}\right)\\
&= \left\{\left\langle\sigma_i', \bigcup_{\substack{j\in I\\ \sigma_j \twoheadrightarrow \sigma_i'}} T_j \cup \{\sigma_j\}\right\rangle \mid \sigma_i \twoheadrightarrow \sigma_i',\ i \in I\right\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_1 \circ [\![\mathsf{post}]\!]_1 = [\![\mathsf{post}]\!]_A^{\sharp 1} \circ \alpha_1$. Consider $c_1 = \{\langle \sigma_i, T_i \rangle \mid i \in I \subseteq \mathbb{N}\}$, then:

$$
\begin{aligned}
(\alpha_1 \circ [\![\mathsf{post}]\!])(c_1) &= \alpha_1 \left(\{\langle \sigma_i', T_i \cup \{\sigma_i\}\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i'\}\right) \\
&= \left\{ \left\langle \sigma_i',\ \bigcup_{\substack{j \in I \\ \sigma_j \twoheadrightarrow \sigma_i'}} T_j \cup \{\sigma_j\} \right\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i' \right\} \\
[\![\mathsf{post}]\!]_A^{\sharp 1} \circ \alpha_1(c_1) &= [\![\mathsf{post1}]\!]_A^{\sharp} \left(\left\{ \left\langle \sigma_i,\ \bigcup_{\substack{h \in I \\ \sigma_h = \sigma_i}} T_h \right\rangle \mid i \in I \right\} \right) \\
&= \left\{ \left\langle \sigma_i',\ \bigcup_{\substack{j \in I \\ \sigma_j \twoheadrightarrow \sigma_i'}} \left(\bigcup_{\substack{h \in I \\ \sigma_h = \sigma_i}} T_h \right) \cup \{\sigma_j\} \right\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i' \right\} \\
&= \left\{ \left\langle \sigma_i',\ \bigcup_{\substack{h \in I \\ \sigma_h \twoheadrightarrow \sigma_i'}} T_h \cup \{\sigma_h\} \right\rangle \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma_i' \right\}
\end{aligned}
$$

where we use the fact that if $\sigma_h = \sigma_j$ and $\sigma_j \twoheadrightarrow \sigma_i'$ then obviously $\sigma_h \twoheadrightarrow \sigma_i'$. Thus we can conclude that the domain is globally complete for post.

We present a small example to better visualise what the discussed $[\![\mathsf{post}]\!]_A^{\sharp}$ does.

Example 4.12. Consider the transition system in Figure 4.1, and consider the state $c = \{\langle \mathsf{c}, \{\mathsf{a}\}\rangle, \langle \mathsf{d}, \{\mathsf{b}\}\rangle\}$, we have that $\alpha_1(c) = c$.

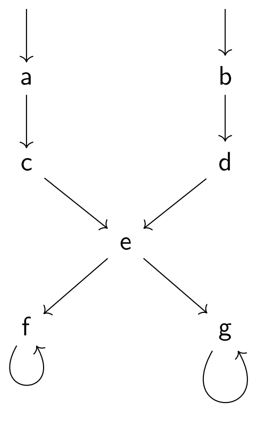

Figure 4.1: An example to show how $[\![\mathsf{post}]\!]_A^{\sharp}$ works.

Now

$$
\begin{aligned}
[\![\mathsf{post}]\!]_A^{\sharp} \circ \alpha_1(c) &= [\![\mathsf{post}]\!]_A^{\sharp}(c) = \{\langle \mathsf{e}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}\rangle\} \\
\alpha_1 \circ [\![\mathsf{post}]\!](c) &= \alpha_1 \{\langle \mathsf{e}, \{\mathsf{a}, \mathsf{c}\}\rangle, \langle \mathsf{e}, \{\mathsf{b}, \mathsf{d}\}\rangle\} = \{\langle \mathsf{e}, \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}\rangle\}
\end{aligned}
$$

Then consider the state $d = \{\langle \mathsf{e}, \{\mathsf{a}, \mathsf{c}\}\rangle, \langle \mathsf{e}, \{\mathsf{b}, \mathsf{d}\}\rangle\}$ and apply again the post operation. Then:

$$\begin{aligned}
[\![\mathsf{post}]\!]^\sharp_A \circ \alpha_1(d) &= [\![\mathsf{post}]\!]^\sharp_A(\{\langle \mathsf{e}, \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}\rangle\}) = \{\langle \mathsf{f}, \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}\rangle, \langle \mathsf{g}, \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}\rangle\} \\
\alpha_1 \circ [\![\mathsf{post}]\!](d) &= \alpha_1 \left(\{\langle \mathsf{f}, \{\mathsf{a},\mathsf{c},\mathsf{e}\}\rangle, \langle \mathsf{g}, \{\mathsf{a},\mathsf{c},\mathsf{e}\}\rangle, \langle \mathsf{f}, \{\mathsf{b},\mathsf{d},\mathsf{e}\}\rangle, \langle \mathsf{g}, \{\mathsf{b},\mathsf{d},\mathsf{e}\}\rangle\}\right) \\
&= \{\langle \mathsf{f}, \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}\rangle, \langle \mathsf{g}, \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}\rangle\}
\end{aligned}$$

$[\![\mathsf{post}]\!]^\sharp_A$ computation, case $a \in \mathsf{C}^n$ Consider $a = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$ such that $\alpha_n(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{post}]\!]^\sharp_A(a) &= (\alpha_n [\![\mathsf{post}]\!] \gamma_n)(a) = (\alpha_n [\![\mathsf{post}]\!] \gamma_n)\left(\left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}\right) \\
&= (\alpha_n [\![\mathsf{post}]\!]) \left(\{\langle \sigma_i, T'_i\rangle :: S'_i \mid i \in I,\ T'_i \subseteq T_i,\ S'_i \in \gamma_{n-1}(\{S_i\})\}\right) \\
&= \alpha_n \left(\{\langle \sigma'_i, T'_i \cup \{\sigma_i\}\rangle \mid i \in I,\ T'_i \subseteq T_i,\ \sigma_i \twoheadrightarrow \sigma'_i, S'_i \in \gamma_{n-1}(\{S_i\})\}\right) \\
&= \left\{ \left\langle \sigma'_i, \bigcup_{\substack{j \in I \\ \sigma_j \twoheadrightarrow \sigma'_i}} T_j \cup \{\sigma_j\} \right\rangle :: S''_i \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i, \right. \\
&\qquad\qquad \left. S''_i \in \operatorname{pop}\left(\alpha_{n-1}\left(\{\langle \sigma'_i, T_i \cup \{\sigma_i\}\rangle :: S_i \mid i \in I, \sigma_i \twoheadrightarrow \sigma'_i\}\right)\right)\right\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_n \circ [\![\mathsf{post}]\!]_n = [\![\mathsf{post}]\!]^{\sharp n}_A \circ \alpha_n$. Consider $c_n = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$, then:

$$\begin{aligned}
(\alpha_n \circ [\![\mathsf{post}]\!])(c_n) &= \alpha_n \left(\overbrace{\left\{\langle \sigma'_i, T_i \cup \{\sigma_i\}\rangle :: S_i^{(n-1)} \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i\right\}}^{c'_n} \right) \\
&= \mathrm{m}_n \left(\{\langle \sigma'_i, T_i \cup \{\sigma_i\}\rangle :: S'_i \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i,\ S'_i \in \operatorname{pop}(\alpha_{n-1}(c'_n))\}\right) \\
&= \left\{ \left\langle \sigma'_i, \bigcup_{\substack{j \in I \\ \sigma'_j = \sigma'_i}} T_j \cup \{\sigma_j\} \right\rangle :: S'_i \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i,\ S'_i \in \operatorname{pop}(\alpha_{n-1}(c'_n)) \right\} \\
[\![\mathsf{post}]\!]^\sharp_A \circ \alpha_n(c_n) &= [\![\mathsf{post}]\!]^\sharp_A \left(\left\{ \left\langle \sigma_i, \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\rangle :: S'_i \mid i \in I,\ S'_i \in \operatorname{pop}(\alpha_{n-1}(c_n)) \right\} \right) \\
&= \left\{ \left\langle \sigma'_i, \bigcup_{\substack{h \in I \\ \sigma_h \twoheadrightarrow \sigma'_i}} \left(\bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right) \cup \{\sigma_h\} \right\rangle :: S''_i \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i, \right. \\
&\quad \left. S''_i \in \operatorname{pop}\left(\alpha_{n-1} \left(\left\{ \left\langle \sigma'_i, \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \cup \{\sigma_j\} \right\rangle :: S'_i \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i,\ S'_i \in \operatorname{pop}(\alpha_{n-1}(c_n)) \right\} \right) \right) \right\} \\
&= \left\{ \left\langle \sigma'_i, \bigcup_{\substack{h \in I \\ \sigma_h \twoheadrightarrow \sigma'_i}} T_h \cup \{\sigma_h\} \right\rangle :: S''_i \mid i \in I,\ \sigma_i \twoheadrightarrow \sigma'_i, S''_i \in \operatorname{pop}(\alpha_{n-1}(c'_n)) \right\}
\end{aligned}$$

where we use the fact that if $\sigma_h = \sigma_j$ and $\sigma_j \twoheadrightarrow \sigma'_i$ then obviously $\sigma_h \twoheadrightarrow \sigma'_i$. As in the case of $[\![\mathsf{next}]\!]^\sharp_A$ the last passage is justified from the fact that ultimately the abstraction is applied to the set of stacks *after* the post has been computed, and it is of no influence the fact that in the second case $S_i' \in \text{pop}(\alpha_{n-1}(c_n))$, since if any unification occurred at that first level, it will happen again in $S_i'' \in \text{pop}(\alpha_{n-1}(c_n'))$ for the same reason that f $\sigma_h = \sigma_j$ and $\sigma_j \twoheadrightarrow \sigma_i'$ then obviously $\sigma_h \twoheadrightarrow \sigma_i'$. Thus we can conclude that the domain is globally complete for post.

$[\![\mathsf{push}]\!]_A^\sharp$ **computation** Consider $a = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$ such that $\alpha_n(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{push}]\!]_A^\sharp(a) &= (\alpha_{n+1}[\![\mathsf{push}]\!]\gamma_n)(a) = (\alpha_n[\![\mathsf{push}]\!]\gamma_n)\left(\left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}\right)\\
&= (\alpha_{n+1}[\![\mathsf{push}]\!])\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \alpha_{n+1}\left(\{\langle\sigma_i, \varnothing\rangle :: \langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \{\langle\sigma_i, \varnothing\rangle :: \langle\sigma_i, T_i\rangle :: S_i \mid i \in I\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_{n+1} \circ [\![\mathsf{push}]\!]_n = [\![\mathsf{push}]\!]_A^{\sharp n} \circ \alpha_n$.

Consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha_{n+1} \circ [\![\mathsf{push}]\!])(c_n) &= \alpha_{n+1}\left(\{\langle\sigma_i, \varnothing\rangle :: \langle\sigma_i, T_i\rangle :: S_i \mid i \in I\}\right)\\
&= \left\{\langle\sigma_i, \varnothing\rangle :: \left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \text{pop}(\alpha_{n-1}(c_n))\right\}\\
[\![\mathsf{push}]\!]_A^\sharp \circ \alpha_n(c_n) &= [\![\mathsf{push}]\!]_A^\sharp\left(\left\{\left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \text{pop}(\alpha_{n-1}(c_n))\right\}\right)\\
&= \left\{\langle\sigma_i, \varnothing\rangle :: \left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \text{pop}(\alpha_{n-1}(c))\right\}
\end{aligned}$$

Thus we can conclude that the domain is globally complete for push.

$[\![\mathsf{pop}]\!]_A^\sharp$ **computation** Consider $a = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}$ such that $\alpha_n(a) = a$. Then we have:

$$\begin{aligned}
[\![\mathsf{pop}]\!]_A^\sharp(a) &= (\alpha_{n-1}[\![\mathsf{pop}]\!]\gamma_n)(a) = (\alpha_n[\![\mathsf{push}]\!]\gamma_n)\left(\left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\}\right)\\
&= (\alpha_{n-1}[\![\mathsf{pop}]\!])\left(\{\langle\sigma_i, T_i'\rangle :: S_i' \mid i \in I,\ T_i' \subseteq T_i,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \alpha_{n-1}\left(\{S_i' \mid i \in I,\ S_i' \in \gamma_{n-1}(\{S_i\})\}\right)\\
&= \{S_i \mid i \in I\}
\end{aligned}$$

Now to prove completeness we must show that $\alpha_{n-1} \circ [\![\mathsf{pop}]\!]_n = [\![\mathsf{pop}]\!]_A^{\sharp n} \circ \alpha_n$.

Consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha_{n-1} \circ [\![\mathsf{pop}]\!])(c_n) &= \alpha_{n-1}\left(\{S_i \mid i \in I\}\right) = \{S_i' \mid i \in I,\ S_i' \in \text{pop}(\alpha_{n-1}(c_n))\} = \alpha_{n-1}(\text{pop}(c_n))\\
[\![\mathsf{pop}]\!]_A^\sharp \circ \alpha_n(c_n) &= [\![\mathsf{pop}]\!]_A^\sharp\left(\left\{\left\langle\sigma_i, \bigcup_{\substack{j\in I\\ \sigma_j=\sigma_i}} T_j\right\rangle :: S_i' \mid i \in I,\ S_i' \in \text{pop}(\alpha_{n-1}(c_n))\right\}\right)\\
&= \{S_i' \mid i \in I,\ S_i' \in \text{pop}(\alpha_{n-1}(c_n))\} = \alpha_{n-1}(\text{pop}(c_n))
\end{aligned}$$

Thus we can conclude that the domain is globally complete for pop.

4.1.3 A general result

We are interested in this abstraction because it is globally complete on our domain and thanks to the following result it can be used to simplify calculations in the abstract domains we will present in the following. In fact, as we formally proved below, under some conditions that will be systematically satisfied in our setting, it preserves the (local) completeness of other abstractions: this simple but useful result is, to the best of our knowledge, inedited.

Proposition 4.13 (Preservation of local completeness)**.** Let C be a concrete domain, let $A_1, A_2 \in \mathrm{Abs}(C)$, let $c \in C$ and $\mathsf{op} : C \to C$ be an operation on C. Assume the following hold:

1. A_1 is locally complete for op on c;
2. A_2 is globally complete for op;
3. $A_2 \circ A_1 = A_2 \circ A_1 \circ A_2$.

Then $A_2 \circ A_1$ is locally complete or op on c.

Proof. We need to show that

$$(A_2 \circ A_1 \circ \mathsf{op} \circ A_2 \circ A_1)(c) = (A_2 \circ A_1 \circ \mathsf{op})(c)$$

We have that

$$\begin{aligned}(A_2 \circ A_1 \circ \mathsf{op} \circ A_2 \circ A_1)(c) &\overset{3.}{=} (A_2 \circ A_1 \circ A_2 \circ \mathsf{op} \circ A_2 \circ A_1)(c)\\ &\overset{2.}{=} (A_2 \circ A_1 \circ A_2 \circ \mathsf{op} \circ A_1)(c)\\ &\overset{3.}{=} (A_2 \circ A_1 \circ \mathsf{op} \circ A_1)(c)\\ &\overset{1.}{=} (A_2 \circ A_1 \circ \mathsf{op})(c)\end{aligned}$$

□

4.2 Partition-based abstraction

We now consider a first way of lifting a given abstraction on states to an abstraction that acts on stacks: this lifting considers partitioning abstractions, i.e. abstractions that identify elements of the concrete domain. The construction of this abstract domain is the same as the construction of the concrete domain, using the abstracted transition system (i.e. the transition system in which the states are *equivalence classes* of states). An abstract element then is a *set* of abstract stacks.

Definition 4.14 (Equivalence relation)**.** Let X be a non-empty set. We say that $\sim \subseteq X \times X$ is an **equivalence relation** if $\sim$ is reflexive, symmetric and transitive i.e. for all $x, y, z \in X$

- $x \sim x$ (reflexivity);
- if $x \sim y$ then $y \sim x$ (symmetry);

- if $x \sim y$ and $y \sim z$ then $x \sim z$ (transitivity).

Given $x \in X$, we define the **equivalence class** of x with respect to $\sim$ as $[x]_\sim = \{y \mid y \sim x\}$. We define the **quotient** of X with respect to $\sim$, written $X_{/\sim}$ as the set of all equivalence classes: $X_{/\sim} = \{[x]_\sim \mid x \in X\}$. The function

$$\pi_\sim : X \to X_{/\sim} \qquad\qquad \pi_\sim : x \mapsto [x]_\sim$$

is called **projection**.

Definition 4.15 (Partition)**.** Let X be a non-empty set, and let $\mathcal{P} \subseteq 2^X$. We say that $\mathcal{P}$ is a **partition** of X if the following conditions hold:

- $\varnothing \notin \mathcal{P}$;
- $\bigcup_{Y \in \mathcal{P}} = X$;
- for all $Y, Z \in \mathcal{P}$ we have $Y \cap Z = \varnothing$.

An equivalence relation $\sim$ on a set X induces a partition on X by means of the equivalence classes: the collection $X_{/\sim} = \{[x]_\sim \mid x \in X\}$ is a partition. The set $X_{/\sim}$ is the *quotient* of X with respect to $\sim$, and we can think of it as a set in its own right, in which we choose an element to represent its equivalence class. The converse is also true, i.e. given a partition $\mathcal{P}$ we can define the induced equivalence relation $\sim_\mathcal{P}$ by defining $x \sim_\mathcal{P} y$ if and only if x, y belong to the same $X \in \mathcal{P}$.

In our setting, we start from an abstraction on states that identifies some of those states, thus inducing an equivalence relation $\sim$. We want to illustrate how to lift this abstraction to the stacks, and we do so by first lifting the equivalence to *subsets* of states.

Definition 4.16 (Collecting projection)**.** Let X be a non-empty set and let $\sim$ be an equivalence on X. Given the projection $\pi_\sim : X \to X_{/\sim}$ we define the **collecting projection** $\overline{\pi}_\sim : 2^\Sigma \to 2^{X_{/\sim}}$ as the additive lifting of $\pi_\sim$ to the powersets as

$$\overline{\pi}_\sim(Y) = \{\pi_\sim(x) \mid x \in Y\} = \{[x]_\sim \mid x \in Y\}$$

The intuition of this construction is given by Figure4.2.

$$\begin{tikzcd}
X \arrow[r, "\pi_\sim"] \arrow[d, "2^{\cdot}"] & X_{/\sim} \arrow[d, "2^{\cdot}"] \\
2^X \arrow[r, "\overline{\pi}_\sim"] & 2^{X_{/\sim}}
\end{tikzcd}$$

Figure 4.2: Diagram for projection, collecting projection.

Definition 4.17 (Transition relation between equivalence classes)**.** Given a transition system $(\Sigma, I, \rightarrowtriangle)$ with an equivalence relation $\sim$ on Σ, we define the **transition relation on equivalence classes** $\rightarrowtriangle_\sim$ saying that $[\sigma]_\sim \rightarrowtriangle_\sim [\sigma']_\sim$ if $\sigma \rightarrowtriangle \sigma'$.

The previous definition is existential, in the sense that we have a transition between two classes if there exist at least a transition between any two elements of these classes (as representatives can be chosen freely in the class).

4.2.1 Galois Connection

In this first way of lifting the abstraction $(\alpha_\Sigma, \Sigma, A_\Sigma, \gamma_\Sigma)$ where $A_\Sigma = \Sigma_{/\sim}$, i.e. the abstraction on states induces a partition, we replicate the same construction we gave for the abstract domain on a new transition system $(\Sigma_{/\sim}, \overline{\pi}_\sim(I), \twoheadrightarrow_\sim)$, hence a *path* in the abstract domain is a pair $\left\langle [\sigma]_\sim, \tilde{T} \right\rangle \in \Sigma_{/\sim} \times 2^{\Sigma_{/\sim}}$, $\mathrm{P}_{\Sigma_{/\sim}}$ is the set of paths, and stacks are elements of

$$\mathrm{Stacks}_{\Sigma_{/\sim}} = \left(\mathrm{P}_{\Sigma_{/\sim}}\right)^* = \bigcup_n \left(\mathrm{P}_{\Sigma_{/\sim}}\right)^n$$

An abstract stack $\tilde{S}^{n+1} \in \left(\mathrm{P}_{\Sigma_{/\sim}}\right)^{n+1}$ is thus of the form

$$\tilde{S}^{n+1} = \left\langle [\sigma]_\sim, \tilde{T} \right\rangle :: \tilde{S}^n$$

where $[\sigma]_{\sim j} \in \Sigma_{/\sim}$ is an abstract state, $\tilde{T}_j = \{[\sigma']_\sim \mid \sigma' \in T \subseteq \Sigma\} \in 2^{\Sigma_{/\sim}}$ is a set of *abstract* traversed state and $\tilde{S}^n \in \left(\mathrm{P}_{\Sigma_{/\sim}}\right)^n$.

Then we can define the abstract domain and abstract elements as follows, recalling Definition 3.3.

Definition 4.18 (Abstract domain, abstract element)**.** The **abstract domain** is

$$\mathsf{A} = \bigcup_n 2^{\left(\mathrm{P}_{\Sigma_{/\sim}}\right)^n}$$

We will denote with A^n the set of abstract stacks of fixed length n, i.e. $\mathsf{A}^n = 2^{\left(\mathrm{P}_{\Sigma_{/\sim}}\right)^n}$.

An **abstract element** is a finite set of abstract stacks of the same length

$$a_n = \left\{ \left\langle [\sigma]_{\sim j}, \tilde{T}_j \right\rangle :: \tilde{S}_j^{(n-1)} \mid j \in J \subseteq \mathbb{N} \right\}$$

where J is a finite set of indices.

We now define the abstraction α and the concretization γ on a single stack by induction on the length n of the stack, and then we lift it to its collecting version.

Definition 4.19 (Abstraction map)**.** The **abstraction map** α_n is inductively defined as

$$\alpha_1(\langle \sigma, T \rangle) = \langle [\sigma]_\sim, \{[\sigma']_\sim \mid \sigma' \in T\} \rangle = \langle \pi_\sim(\sigma), \overline{\pi}_\sim(T) \rangle$$
$$\alpha_n(\langle \sigma, T \rangle :: S^{n-1}) = \alpha_1(\langle \sigma, T \rangle) :: \alpha_{n-1}(S^{n-1})$$

which gets lifted to $\alpha_n : \mathsf{C}^n \to \mathsf{A}^n$ as

$$\alpha_n \left(\left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\} \right) = \bigcup_{i \in I} \left\{ \alpha_n \left(\langle \sigma_i, T_i \rangle :: S_i^{n-1} \right) \right\}$$

Definition 4.20 (Concretization map)**.** The **concretization map** γ_n is inductively defined as

$$\gamma_1 \left(\left\langle [\sigma]_\sim, \tilde{T} \right\rangle \right) = \left\{ \langle \sigma', T' \rangle \mid \sigma' \sim \sigma,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T} \right\}$$
$$\gamma_n \left(\left\langle [\sigma]_\sim, \tilde{T} :: \tilde{S}^{(n-1)} \right\rangle \right) = \left\{ \langle \sigma', T' \rangle :: S' \mid \sigma' \sim \sigma,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)}) \right\}$$

which gets lifted to $\gamma_n : \mathsf{C}^n \to \mathsf{A}^n$ as

$$\gamma_n \left(\left\{ \left\langle [\sigma]_{\sim j}, \tilde{T}_j \right\rangle :: \tilde{S}_j^{(n-1)} \mid j \in J \subseteq \mathbb{N} \right\} \right) = \bigcup_{j \in J} \gamma_n \left(\left\langle [\sigma]_{\sim j}, \tilde{T}_j \right\rangle :: \tilde{S}_j^{(n-1)} \right)$$

Proposition 4.21. The above construction $(\alpha, \mathsf{C}, \mathsf{A}, \gamma)$ is a Galois Connection.

Proof. We give the proof by induction on the length n of the stacks, by showing that each $\gamma_n \circ \alpha_n$ is a closure. First let $c = \{\langle \sigma_i, T_i \rangle :: S_i \mid i \in I\}$, then we explicitly compute

$$\alpha_n(c) = \left\{ \left\langle [\sigma]_\sim, \overbrace{\{[\sigma']_\sim \mid \sigma' \in T_i\}}^{\overline{\pi}(T_i)} \right\rangle :: \alpha_{n-1}(S_i) \mid i \in I \right\}$$

$$(\gamma_n \circ \alpha_n)(c) = \left\{ \langle \sigma_i', T_i' \rangle :: S_i' \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i)),\ \overline{\pi}(T_i') = \overline{\pi}(T_i),\ S_i' \in \gamma_{n-1}(\alpha_{n-1}(S_i)) \right\}$$

which will be useful in the following.

$(n = 1)$ **(Monotone)** Let $c_1, c_2 \in \mathsf{C}^1$ be such that $c_1 \leq_{\mathsf{C}} c_2$, where

$$c_1 = \{\langle \sigma_i, T_i \rangle \mid i \in I\} \qquad c_2 = \{\langle \sigma_j, T_j \rangle \mid j \in J\}$$

and $c_1 \leq_{\mathsf{C}} c_2$ means that for all $\langle \sigma, T \rangle \in c_1$ we have $\langle \sigma, T \rangle \in c_2$. Now by definition

$$(\gamma_1 \circ \alpha_1)(c_1) = \left\{ \langle \sigma_i', T_i' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i)),\ \overline{\pi}(T_i') = \overline{\pi}(T_i) \right\}$$
$$(\gamma_1 \circ \alpha_1)(c_2) = \left\{ \langle \sigma_j', T_j' \rangle \mid j \in J,\ \sigma_j' \sim \sigma_j,\ T_j' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_j)),\ \overline{\pi}(T_j') = \overline{\pi}(T_j) \right\}$$

Now let $\langle \sigma, T \rangle \in (\gamma_1 \circ \alpha_1)(c_1)$, then there exists $i \in I$ such that $\langle \sigma_i, T_i \rangle \in c_1$, $\sigma \sim \sigma_i$, $T \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i))$, $\overline{\pi}(T) = \overline{\pi}(T_i)$. Then, since $c_1 \leq_{\mathsf{C}} c_2$, we have $\langle \sigma_i, T_i \rangle \in c_2$, so there exists $j \in J$ such that $\langle \sigma_i, T_i \rangle = \langle \sigma_j, T_j \rangle$ and $\sigma \sim \sigma_j$, $T \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_j))$, $\overline{\pi}(T) = \overline{\pi}(T_j)$, hence $\langle \sigma, T \rangle \in (\gamma_1 \circ \alpha_1)(c_2)$.

(Idempotent) Let $c = c_1$ of the previous computation, then we can explicitly compute $(\alpha_1 \circ \gamma_1 \circ \alpha_1)(c)$ as:

$$\begin{aligned}(\alpha_1 \circ \gamma_1 \circ \alpha_1)(c) &= \left\{ \langle [\sigma_i]_\sim, \overline{\pi}(T_i') \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i, T_i' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i)),\ \overline{\pi}(T_i') = \overline{\pi}(T_i) \right\} \\ &= \{\langle [\sigma]_\sim, \overline{\pi}(T_i) \rangle \mid i \in I\} = \alpha_1(c)\end{aligned}$$

since $[\sigma_i']_\sim = [\sigma_i]_\sim$ and $\overline{\pi}(T_i') = \overline{\pi}(T_i)$. Then since $(\alpha_1 \circ \gamma_1 \circ \alpha_1)(c) = \alpha_1(c)$, it follows that $(\gamma_1 \circ \alpha_1)^2(c) = (\gamma_1 \circ \alpha_1 \circ \gamma_1 \circ \alpha_1)(c) = (\gamma_1 \circ \alpha_1)(c)$.

(Extensive) Let $\langle \sigma, T \rangle \in c$, then $\langle \sigma, T \rangle \in (\gamma_1 \circ \alpha_1)(c)$ since $\sigma \sim \sigma$ and $T \subseteq \overline{\pi}^{-1}(\overline{\pi}(T))$ with obviously $\overline{\pi}(T) = \overline{\pi}(T)$.

$(n - 1 \Rightarrow n)$ **(Monotone)** Let $c_1, c_2 \in \mathsf{C}^n$ be such that $c_1 \leq_{\mathsf{C}} c_2$, where

$$c_1 = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\} \qquad c_2 = \left\{ \langle \sigma_j, T_j \rangle :: S_j^{(n-1)} \mid j \in J \right\}$$

and $c_1 \leq_{\mathsf{C}} c_2$ means that for all $\langle \sigma, T \rangle :: S \in c_1$ we have $\langle \sigma, T \rangle :: S \in c_2$. Now by definition

$$(\gamma_n \circ \alpha_n)(c_1) = \left\{ \langle \sigma_i', T_i' \rangle :: S_i' \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i)),\ \overline{\pi}(T_i') = \overline{\pi}(T_i),\ S_i' \in \gamma_{n-1}(\alpha_{n-1}(S_i)) \right\}$$
$$(\gamma_n \circ \alpha_n)(c_2) = \left\{ \langle \sigma_j', T_j' \rangle :: S_j' \mid j \in J,\ \sigma_j' \sim \sigma_j,\ T_j' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_j)),\ \overline{\pi}(T_j') = \overline{\pi}(T_j),\ S_j' \in \gamma_{n-1}(\alpha_{n-1}(S_j)) \right\}$$

Now let $\langle\sigma, T\rangle :: S \in (\gamma_n \circ \alpha_n)(c_1)$, then there exists $i \in I$ such that $\langle\sigma_i, T_i\rangle :: S_i \in c_1$, $\sigma \sim \sigma_i$, $T \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i))$, $\overline{\pi}(T) = \overline{\pi}(T_i)$ and $S \in \gamma_{n-1}(\alpha_{n-1}(S_i))$. Then, since $c_1 \leq_{\mathsf{C}} c_2$, we have $\langle\sigma_i, T_i\rangle :: S_i \in c_2$, so there exists $j \in J$ such that $\langle\sigma_i, T_i\rangle :: S_i = \langle\sigma_j, T_j\rangle :: S_j$ and $\sigma \sim \sigma_j$, $T \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_j))$, $\overline{\pi}(T) = \overline{\pi}(T_j)$, $S \in \gamma_{n-1}(\alpha_{n-1}(S_j))$, hence $\langle\sigma, T\rangle :: S \in (\gamma_1 \circ \alpha_1)(c_2)$.

(Idempotent) Let $c = c_1$ of the previous computation, then we can explicitly compute $(\alpha_n \circ \gamma_n \circ \alpha_n)(c)$ as:

$$\begin{aligned}
(\alpha_n \circ \gamma_n \circ \alpha_n)(c) &= \big\{\langle[\sigma_i]_\sim, \overline{\pi}(T_i')\rangle :: \alpha_{n-1}(S_i') \mid i \in I,\ \sigma_i' \sim \sigma_i, T_i' \subseteq \overline{\pi}^{-1}(\overline{\pi}(T_i)),\\
&\qquad \overline{\pi}(T_i') = \overline{\pi}(T_i),\ S_i' \in \gamma_{n-1}(\alpha_{n-1}(S_i))\big\}\\
&= \{\langle[\sigma]_\sim, \overline{\pi}(T_i) :: \alpha_{n-1}(S_i)\rangle \mid i \in I\} = \alpha_n(c)
\end{aligned}$$

since $[\sigma_i']_\sim = [\sigma_i]_\sim$, $\overline{\pi}(T_i') = \overline{\pi}(T_i)$ and $\alpha_{n-1}(S_i') = \alpha_{n-1}(S_i) = (\alpha_{n-1} \circ \gamma_{n-1} \circ \alpha_{n-1})(S_i)$ for all $S_i' \in (\gamma_{n-1} \circ \alpha_{n-1})(S_i)$ since $\gamma_{n-1} \circ \alpha_{n-1}$ is a closure by inductive hypothesis. Then since $(\alpha_n \circ \gamma_n \circ \alpha_n)(c) = \alpha_n(c)$, it follows that $(\gamma_n \circ \alpha_n)^2(c) = (\gamma_n \circ \alpha_n \circ \gamma_n \circ \alpha_n)(c) = (\gamma_n \circ \alpha_n)(c)$.

(Extensive) Let $\langle\sigma, T\rangle : S \in c$, then $\langle\sigma, T\rangle :: S \in (\gamma_n \circ \alpha_n)(c)$ since $\sigma \sim \sigma$, $T \subseteq \overline{\pi}^{-1}(\overline{\pi}(T))$ with $\overline{\pi}(T) = \overline{\pi}(T)$ and $S \in (\gamma_{n-1} \circ \alpha_{n-1})(S)$ since $\gamma_{n-1} \circ \alpha_{n-1}$ is a closure by inductive hypothesis. □

4.2.2 Abstract Semantics

We now compute the Best Correct Approximations for basic expressions and study under which conditions on concrete elements we have local completeness. We will see that for the push and pop operators we actually have global completeness.

$[\![\mathsf{p}?]\!]^\sharp_A$ computation Consider $a = \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}$ (not collecting semantics). Then we have:

$$\begin{aligned}
[\![\mathsf{p}?]\!]^\sharp_A(a) &= (\alpha[\![\mathsf{p}?]\!]\gamma)(a) = (\alpha[\![\mathsf{p}?]\!]\gamma)\left(\left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\right)\\
&= (\alpha[\![\mathsf{p}?]\!])\left(\left\{\langle\sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \alpha\left(\left\{\langle\sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ \sigma \models \mathsf{p},\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \alpha\left(\left\{\langle\sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim \cap \mathsf{p},\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \begin{cases} \bot & \text{if } \forall\sigma' \in [\sigma]_\sim\ \ (\!|\mathsf{p}?|\!)\,\langle\sigma', T\rangle = \varnothing\\ \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)} & \text{if } \exists\sigma' \in [\sigma]_\sim\ \ (\!|\mathsf{p}?|\!)\,\langle\sigma', T\rangle \neq \varnothing \end{cases}
\end{aligned}$$

For this and all the following abstract operations, the collecting semantics is obtained by just taking the union of the single computations, i.e.

$$[\![\mathsf{p}?]\!]^\sharp_A\left(\left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i \mid i \in I\right\}\right) = \bigcup_{i \in I}\left\{[\![\mathsf{p}?]\!]^\sharp_A\left(\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i\right)\right\}$$

In order to have local completeness, we must check under which conditions $\alpha_n \circ [\![\mathsf{p}?]\!]_n = [\![\mathsf{p}?]\!]_A^{\sharp n} \circ \alpha_n$. Consider $c_n = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha \circ [\![\mathsf{p}?]\!])(c_n) &= \alpha\left(\left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I,\ \sigma_i \models \mathsf{p}\right\}\right)\\
&= \left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I,\ \sigma_i \models \mathsf{p}\right\}\\
([\![\mathsf{p}?]\!]_A^\sharp \circ \alpha)(c_n) &= [\![\mathsf{p}?]\!]_A^\sharp\left(\left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}\right)\\
&= \left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I,\ \exists \sigma' \sim \sigma,\ \sigma' \models \mathsf{p}\right\}
\end{aligned}$$

Hence completeness holds if either $\sigma \models \mathsf{p}$ or for all $\sigma' \in [\sigma]_\sim$ $\sigma' \not\models \mathsf{p}$. In other words, the problem arises if $\sigma \not\models \mathsf{p}$ and there exists $\sigma' \sim \sigma$ such that $\sigma' \models \mathsf{p}$. To avoid this, our partition $\sim$ must be a refinement of the partition $\{\mathsf{p}, \neg\mathsf{p}\}$.

$[\![\mathsf{loop}?]\!]_A^\sharp$ **computation** Consider $a = \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}$ (not collecting semantics). Then we have:

$$\begin{aligned}
[\![\mathsf{p}?]\!]_A^\sharp(a) &= (\alpha[\![\mathsf{loop}?]\!]\gamma)(a) = (\alpha[\![\mathsf{loop}?]\!]\gamma)\left(\left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\right)\\
&= (\alpha[\![\mathsf{loop}?]\!])\left(\left\{\langle \sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \alpha\left(\left\{\langle \sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ \sigma' \in T',\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \begin{cases} \bot & \text{if } \forall \sigma' \in [\sigma]_\sim,\ \sigma' \notin \overline{\pi}_\sim^{-1}(\tilde{T})\\ \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)} & \text{if } \exists \sigma' \in [\sigma]_\sim,\ \sigma' \in \overline{\pi}_\sim^{-1}(\tilde{T}) \end{cases}
\end{aligned}$$

In order to have local completeness, we must check under which conditions $\alpha_n \circ [\![\mathsf{loop}?]\!]_n = [\![\mathsf{loop}?]\!]_A^{\sharp n} \circ \alpha_n$. Consider $c_n = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha \circ [\![\mathsf{loop}?]\!])(c_n) &= \alpha\left(\left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I,\ \sigma_i \in T_i\right\}\right)\\
&= \left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I,\ \sigma_i \in T_i\right\}\\
([\![\mathsf{loop}?]\!]_A^\sharp \circ \alpha)(c_n) &= [\![\mathsf{loop}?]\!]_A^\sharp\left(\left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}\right)\\
&= \left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I,\ \exists \sigma' \in [\sigma_i]_\sim,\ \sigma' \in \overline{\pi}_\sim^{-1}(\tilde{T}_i)\right\}
\end{aligned}$$

Hence completeness holds if either $\sigma \in T$, so there is a loop, or, if there is no loop, i.e. for all $\sigma' \in [\sigma]_\sim$ we have $\sigma' \notin \{\rho' \in \Sigma \mid \exists \rho \in T,\ \rho' \sim \rho\}$. This means that if $\sigma \notin T$ and $\sigma' \in T$, it must hold that $\sigma \not\sim \sigma'$.

$[\![\mathsf{next}]\!]_A^\sharp$ **computation** Consider $a = \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}$ (not collecting semantics). Then we have:

$$\begin{aligned}
[\![\mathsf{next}]\!]_A^\sharp(a) &= (\alpha[\![\mathsf{next}]\!]\gamma)(a) = (\alpha[\![\mathsf{next}]\!]\gamma)\left(\left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\right) =\\
&= (\alpha[\![\mathsf{next}]\!])\left(\left\{\langle \sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \alpha\left(\left\{\langle \sigma'', \varnothing\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ \sigma' \twoheadrightarrow \sigma'',\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\
&= \left\{\langle [\sigma'']_\sim, \varnothing\rangle :: \tilde{S}^{(n-1)} \mid \exists \sigma' \in [\sigma]_\sim,\ \sigma' \twoheadrightarrow \sigma''\right\}
\end{aligned}$$

In order to have local completeness, we must check under which conditions $\alpha_n \circ [\![\mathsf{post}]\!]_n = [\![\mathsf{post}]\!]_A^{\sharp n} \circ \alpha_n$. Consider $c_n = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}(\alpha \circ [\![\mathsf{next}]\!])(c_n) &= \alpha\left(\left\{\langle \sigma_i', \varnothing\rangle :: S_i^{(n-1)} \mid \sigma_i \twoheadrightarrow \sigma_i',\ i \in I\right\}\right)\\ &= \left\{\langle [\sigma_i']_\sim, \varnothing\rangle :: \tilde{S}_i^{(n-1)} \mid \sigma_i \twoheadrightarrow \sigma_i',\ i \in I\right\}\\ ([\![\mathsf{next}]\!]_A^\sharp \circ \alpha)(c_n) &= [\![\mathsf{next}]\!]_A^\sharp\left(\left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}\right)\\ &= \left\{\langle [\sigma_i']_\sim, \varnothing\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I,\ \exists \sigma'' \in [\sigma_i]_\sim,\ \sigma'' \twoheadrightarrow \sigma_i'\right\}\end{aligned}$$

where σ_i' is any representative of its class. Hence completeness holds if elements equivalent to σ have post-images that are equivalent to post-images of σ.

$[\![\mathsf{post}]\!]_A^\sharp$ computation Consider $a = \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}$ (not collecting). Then we have:

$$\begin{aligned}[\![\mathsf{post}]\!]_A^\sharp(a) &= (\alpha[\![\mathsf{post}]\!]\gamma)(a) = (\alpha[\![\mathsf{post}]\!]\gamma)\left(\left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\right) =\\ &= (\alpha[\![\mathsf{post}]\!])\left(\left\{\langle \sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\ &= \alpha\left(\left\{\langle \sigma'', T' \cup \{\sigma'\}\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ \sigma' \twoheadrightarrow \sigma'',\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\ &= \left\{\left\langle [\sigma'']_\sim, \tilde{T} \cup \{[\sigma]_\sim\}\right\rangle :: \tilde{S}^{(n-1)} \mid \exists \sigma' \in [\sigma]_\sim,\ \sigma' \twoheadrightarrow \sigma''\right\}\end{aligned}$$

In order to have local completeness, we must check under which conditions $\alpha_n \circ [\![\mathsf{post}]\!]_n = [\![\mathsf{post}]\!]_A^{\sharp n} \circ \alpha_n$. Consider $c_n = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}(\alpha \circ [\![\mathsf{post}]\!])(c_n) &= \alpha\left(\left\{\langle \sigma_i', T_i \cup \{\sigma_i\}\rangle :: S_i^{(n-1)} \mid \sigma_i \twoheadrightarrow \sigma_i',\ i \in I\right\}\right)\\ &= \left\{\left\langle [\sigma_i']_\sim, \tilde{T}_i \cup \{[\sigma_i]_\sim\}\right\rangle :: \tilde{S}_i^{(n-1)} \mid \sigma_i \twoheadrightarrow \sigma_i',\ i \in I\right\}\\ ([\![\mathsf{post}]\!]_A^\sharp \circ \alpha)(c_n) &= [\![\mathsf{post}]\!]_A^\sharp\left(\left\{\left\langle [\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}\right)\\ &= \left\{\left\langle [\sigma_i']_\sim, \tilde{T}_i \cup \{[\sigma_i]_\sim\}\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I,\ \exists \sigma'' \in [\sigma_i]_\sim,\ \sigma'' \twoheadrightarrow \sigma_i'\right\}\end{aligned}$$

where σ_i' is any representative of its class. Hence, as in the $[\![\mathsf{next}]\!]_A^\sharp$ case, completeness holds if elements equivalent to σ have post-images that are equivalent to post-images of σ.

$[\![\mathsf{push}]\!]_A^\sharp$ computation Consider $a = \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}$ (not collecting). Then we have:

$$\begin{aligned}[\![\mathsf{push}]\!]_A^\sharp(a) &= (\alpha[\![\mathsf{push}]\!]\gamma)(a) = (\alpha[\![\mathsf{post}]\!]\gamma)\left(\left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\right) =\\ &= (\alpha[\![\mathsf{push}]\!])\left(\left\{\langle \sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\ &= \alpha\left(\left\{\langle \sigma', \varnothing\rangle :: \langle \sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right)\\ &= \langle [\sigma]_\sim, \varnothing\rangle :: \left\langle [\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\end{aligned}$$

Now we show that push is globally complete: consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha \circ [\![\mathsf{push}]\!])(c_n) &= \alpha\left(\left\{\langle\sigma_i, \varnothing\rangle :: \langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid\ i \in I\right\}\right) \\
&= \left\{\langle[\sigma_i]_\sim, \varnothing\rangle :: \left\langle[\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\} \\
([\![\mathsf{push}]\!]^\sharp_A \circ \alpha)(c_n) &= [\![\mathsf{push}]\!]^\sharp_A\left(\left\{\left\langle[\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}\right) \\
&= \left\{\langle[\sigma_i]_\sim, \varnothing\rangle :: \left\langle[\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}
\end{aligned}$$

$[\![\mathsf{pop}]\!]^\sharp_A$ computation Consider $a = \left\langle[\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}$ (not collecting). Then we have:

$$\begin{aligned}
[\![\mathsf{pop}]\!]^\sharp_A(a) &= (\alpha[\![\mathsf{pop}]\!]\gamma)(a) = (\alpha[\![\mathsf{pop}]\!]\gamma)\left(\left\langle[\sigma]_\sim, \tilde{T}\right\rangle :: \tilde{S}^{(n-1)}\right) = \\
&= (\alpha[\![\mathsf{pop}]\!])\left(\left\{\langle\sigma', T'\rangle :: S' \mid \sigma' \in [\sigma]_\sim,\ T' \subseteq \overline{\pi}_\sim^{-1}(\tilde{T}),\ \overline{\pi}(T') = \tilde{T},\ S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right) \\
&= \alpha\left(\left\{S' \mid S' \in \gamma_{n-1}(\tilde{S}^{(n-1)})\right\}\right) \\
&= \tilde{S}^{(n-1)}
\end{aligned}$$

Now we show that pop is globally complete: consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I \subseteq \mathbb{N}\right\}$, then:

$$\begin{aligned}
(\alpha \circ [\![\mathsf{pop}]\!])(c_n) &= \alpha\left(\left\{S_i^{(n-1)} \mid\ i \in I\right\}\right) \\
&= \left\{\tilde{S}_i^{(n-1)} \mid i \in I\right\} \\
([\![\mathsf{pop}]\!]^\sharp_A \circ \alpha)(c_n) &= [\![\mathsf{pop}]\!]^\sharp_A\left(\left\{\left\langle[\sigma_i]_\sim, \tilde{T}_i\right\rangle :: \tilde{S}_i^{(n-1)} \mid i \in I\right\}\right) \\
&= \left\{\tilde{S}_i^{(n-1)} \mid i \in I\right\}
\end{aligned}$$

We now recover two examples mentioned in the previous chapters to show one case in which the abstract domain we choose is not locally complete for the property we wish to verify, and one case in which it is.

Example 4.22. Consider again the transition system introduced in example 3.10:

$$\begin{tikzcd}
{} \arrow[d] & {} \arrow[d] \\
\mathsf{a} \arrow[d] & \mathsf{b} \arrow[l] \\
\mathsf{c} \arrow[loop below]
\end{tikzcd}$$

and consider the following partition that identifies a and b:

$$\mathsf{a}, \mathsf{b} \mapsto \{\mathsf{a}, \mathsf{b}\} = [\mathsf{a}]_\sim \qquad\qquad \mathsf{c} \mapsto \{\mathsf{c}\} = [\mathsf{c}]_\sim$$

The corresponding abstract transition system is as follows:

$$\begin{tikzcd}
{} \arrow[d] \\
{[\mathsf{a}]_\sim} \arrow[loop right] \arrow[d] \\
{[\mathsf{c}]_\sim} \arrow[loop below]
\end{tikzcd}$$

We notice that this introduces a spurious loop of $\{\mathsf{a},\mathsf{b}\}$ on itself: this introduces a spurious error alarm in the computation of the already verified true property $\varphi' = \mathsf{AF}\ \mathsf{c}$. In particular we may look at the proof obligation that fails in the derivation tree of the computation. For simplicity, we omit the push and pop commands from the definition of $\lfloor\overline{\mathsf{AF}\ \mathsf{c}}\rfloor$, since they are not relevant in this discussion. Thus we have the derivation depicted in Figure 4.3, where $p = \{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle\}$, $w = \{\langle \mathsf{a},\{\mathsf{b}\}\rangle\}$.

$$\dfrac{\dfrac{\vdots}{\vdash_A [p]\ (\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?)^*\ [p\cup w]}\ \text{(rec)} \qquad \dfrac{\mathbb{C}^A_{p\cup w}(\mathsf{loop}?)}{\vdash_A [p\cup w]\ \mathsf{loop}?\ [\varnothing]}\ \text{(trsf)}}{\vdash_A [\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle\}]\ ((\mathsf{post}; \{\mathsf{a},\mathsf{b}\}?)^*; \mathsf{loop}?)\ [\varnothing]}\ \text{(seq)}$$

Figure 4.3: Proof tree for the property $\varphi' = \mathsf{AF}\ \mathsf{c}$ for the partition that identifies a,b (Ex. 4.22).

In particular we see that what fails is

$$\dfrac{\mathbb{C}^A_{\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle,\langle \mathsf{a},\{\mathsf{b}\}\rangle\}}(\mathsf{loop}?)}{\vdash_A [\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle, \langle \mathsf{a},\{\mathsf{b}\}\rangle\}]\ \mathsf{loop}?\ [\varnothing]}\ \text{(transfer)}$$

where the proof obligation $\mathbb{C}^A_{\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle,\langle \mathsf{a},\{\mathsf{b}\}\rangle\}}(\mathsf{loop}?)$ required is

$$\gamma\alpha[\![\mathsf{loop}?]\!](\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle, \langle \mathsf{a},\{\mathsf{b}\}\rangle\}) = \gamma\alpha[\![\mathsf{loop}?]\!]\gamma\alpha(\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle, \langle \mathsf{a},\{\mathsf{b}\}\rangle\})$$

which fails because $[\![\mathsf{loop}?]\!](\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle, \langle \mathsf{a},\{\mathsf{b}\}\rangle\}) = \varnothing$, but

$$\begin{aligned}
[\![\mathsf{loop}?]\!]\gamma\alpha(\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle, \langle \mathsf{a},\{\mathsf{b}\}\rangle\}) &= [\![\mathsf{loop}?]\!]\gamma\,\{\langle [\mathsf{a}]_\sim,\varnothing\rangle, \langle [\mathsf{a}]_\sim,\{[\mathsf{a}]_\sim\}\rangle\} \\
&= [\![\mathsf{loop}?]\!]\,\{\langle \mathsf{a},\varnothing\rangle, \langle \mathsf{b},\varnothing\rangle, \langle \mathsf{a},\{\mathsf{a}\}\rangle, \langle \mathsf{a},\{\mathsf{b}\}\rangle, \langle \mathsf{b},\{\mathsf{a}\}\rangle, \\
&\qquad \langle \mathsf{b},\{\mathsf{b}\}\rangle, \langle \mathsf{a},\{\mathsf{a},\mathsf{b}\}\rangle, \langle \mathsf{b},\{\mathsf{a},\mathsf{b}\}\rangle\} \neq \varnothing
\end{aligned}$$

Example 4.23. Consider again the composite transition system introduced in Example 1.21:

$$\begin{tikzcd}
{} \arrow[r] & \mathsf{rs} \arrow[loop above] \arrow[r] & \mathsf{gs} \arrow[loop above] \arrow[r] \arrow[d, bend left] \arrow[dr] & \mathsf{ys} \arrow[loop above] \arrow[ll, bend right] \\
& \mathsf{rd} \arrow[u] \arrow[ur] & \mathsf{gd} \arrow[loop below] \arrow[u, bend left] \arrow[ur] \arrow[r] & \mathsf{yd} \arrow[u] \arrow[ull]
\end{tikzcd}$$

$$\begin{tikzcd} & {} \arrow[d] \\ {[\mathsf{rd}]_\sim} \arrow[r] & {[\neg\mathsf{rd}]_\sim} \arrow[loop right] \end{tikzcd}$$

Figure 4.4: Abstract transition system for the traffic light partitioning $\{\mathsf{rd}, \neg\mathsf{rd}\}$ (Ex. 4.23).

and consider the partition that distinguishes the "bad" state rd from the others, namely $\Sigma_{/\sim} = \{[\mathsf{rd}]_\sim, [\neg\mathsf{rd}]_\sim\}$

$$\alpha_\Sigma : \Sigma \to \Sigma_{/\sim} \qquad \alpha : \mathsf{rd} \mapsto [\mathsf{rd}]_\sim \qquad \alpha : \sigma \mapsto [\neg\mathsf{rd}]_\sim \ \forall \sigma \neq \mathsf{rd}$$

Then the abstract transition system is the one depicted in Figure 4.4. We can see that the abstract domain can successfully prove that the property $\varphi' = \mathsf{AG}\ (\neg\mathsf{rd})$ holds: we show the proof obligation for the property. For simplicity, we omit the push and pop commands from the definition of $\lfloor \overline{\mathsf{AG}\ (\neg\mathsf{rd})} \rfloor$ since they are not relevant in the discussion. Thus we have the derivation depicted in Figure 4.5.

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^A_{\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}}(\mathsf{next})}{\vdash_A [\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}]\ \mathsf{next}\ [\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}]}\,(\text{trsf})}{\vdash_A [\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}]\ \mathsf{next}^*\ [\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}]}\,(\text{it}) \qquad \dfrac{\mathbb{C}^A_{\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}}(\mathsf{rd}?)}{\vdash_A [\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}]\ \mathsf{rd}?\ [\varnothing]}\,(\text{trsf})}{\vdash_A [\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}]\ (\mathsf{next}^*;\mathsf{rd}?)\ [\varnothing]}\,(\text{seq})$$

Figure 4.5: Proof tree for the property $\varphi' = \mathsf{AG}\ (\neg\mathsf{rd})$ for the traffic light partitioning $\{\mathsf{rd}, \neg\mathsf{rd}\}$.

We see that the two proof obligations $\mathbb{C}^A_{\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}}(\mathsf{next})$ and $\mathbb{C}^A_{\{\langle[\neg\mathsf{rd}]_\sim,\varnothing\rangle\}}(\mathsf{rd}?)$ are satisfied because the domain satisfies the conditions studied above: the abstract domain is precisely the partition that distinguishes rd from the rest of states, so both the conditions studied before for the completeness of $\mathsf{next}, \mathsf{rd}?$ hold.

4.2.3 Using the local-completeness preserving abstraction

We may exploit the abstraction seen in the previous section to simplify the use of the partition-based abstractions. In order to exploit Proposition 4.13 we only need to prove that the commutation property $A_2 \circ A_1 \circ A_2 = A_2 \circ A_1$ holds if we let A_2 the abstraction that merges pasts (defined in the previous section) and A_1 any partition-based abstraction.

Proposition 4.24. Let A_1 be the abstraction of Proposition 4.21 and let A_2 be the abstraction of Proposition 4.10. Then we have $A_2 \circ A_1 \circ A_2 = A_2 \circ A_1$.

Proof. We prove the commutation relation in the case $n = 1$. Let us first explicitly describe the two abstractions: let $c = \{\langle \sigma_i, T_i \rangle \mid i \in I\}$, then

$$\begin{aligned}
A_2(c) &= \gamma^2 \circ \alpha^2(c) = \gamma^2 \left(\left\{ \left\langle \sigma_i, \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\rangle \mid i \in I \right\} \right) \\
&= \left\{ \langle \sigma_i, T_i' \rangle \mid i \in I,\ T_i' \subseteq \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \\
A_1(c) &= \gamma^1 \circ \alpha^1(c) = \gamma^1 \left(\{ \langle [\sigma_i]_\sim, \{ [\sigma']_\sim \mid \sigma' \in T_i \} \rangle \mid i \in I \} \right) \\
&= \left\{ \left\langle \sigma_i', \hat{T}_i \right\rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ \hat{T}_i \subseteq \pi^{-1} \left(\{ [\sigma']_\sim \mid \sigma' \in T_i \} \right) \right\}
\end{aligned}$$

Then we may compute $(A_2 \circ A_1)(c)$ and $(A_2 \circ A_1 \circ A_2)(c)$:

$$\begin{aligned}
(A_2 \circ A_1)(c) &= A_2 \left(\left\{ \left\langle \sigma_i', \hat{T}_i \right\rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ \hat{T}_i \subseteq \pi^{-1} \left(\{ [\sigma']_\sim \mid \sigma' \in T_i \} \right) \right\} \right) \\
&= \left\{ \langle \sigma_i', T_i' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i' \subseteq \left(\bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} \hat{T}_j \right),\ \hat{T}_j \subseteq \pi^{-1} \left(\{ [\sigma']_\sim \mid \sigma' \in T_i \} \right) \right\} \\
&= \left\{ \langle \sigma_i', T_i' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i' \subseteq \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} \pi^{-1} \left(\{ [\sigma']_\sim \mid \sigma' \in T_i \} \right) \right\} \\
&= \left\{ \langle \sigma_i', T_i' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i' \subseteq \left\{ \sigma' \mid \sigma' \sim \sigma,\ \sigma \in \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \right\}
\end{aligned}$$

$$\begin{aligned}
(A_2 \circ A_1 \circ A_2)(c) &= (A_2 \circ A_1) \left(\left\{ \langle \sigma_i, T_i' \rangle \mid i \in I,\ T_i' \subseteq \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \right) \\
&= A_2 \left(\left\{ \left\langle \sigma_i', \hat{T}_i \right\rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ \hat{T}_i \subseteq \pi^{-1} \left(\left\{ [\sigma']_\sim \mid \sigma' \in \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \right) \right\} \right) \\
&= \left\{ \langle \sigma_i', T_i'' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i'' \subseteq \left(\bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} \hat{T}_j \right),\ \hat{T}_i \subseteq \pi^{-1} \left(\left\{ [\sigma']_\sim \mid \sigma' \in \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \right) \right\} \\
&= \left\{ \langle \sigma_i', T_i'' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i'' \subseteq \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} \pi^{-1} \left(\left\{ [\sigma']_\sim \mid \sigma' \in \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \right) \right\} \\
&= \left\{ \langle \sigma_i', T_i'' \rangle \mid i \in I,\ \sigma_i' \sim \sigma_i,\ T_i'' \subseteq \left\{ \sigma' \mid \sigma' \sim \sigma,\ \sigma \in \bigcup_{\substack{j \in I \\ \sigma_j = \sigma_i}} T_j \right\} \right\}
\end{aligned}$$

So $(A_2 \circ A_1)(c) = (A_2 \circ A_1 \circ A_2)(c)$, as we wanted. □

We present here a small example to show how the use of the past-merging abstraction helps in the computation.

Example 4.25. Consider again the transition system of Example 4.12:

$$\begin{tikzcd}
{} \arrow[d] & & {} \arrow[d] \\
\mathsf{a} \arrow[d] & & \mathsf{b} \arrow[d] \\
\mathsf{c} \arrow[dr] & & \mathsf{d} \arrow[dl] \\
& \mathsf{e} \arrow[dl] \arrow[dr] & \\
\mathsf{f} \arrow[loop below] & & \mathsf{g} \arrow[loop below]
\end{tikzcd}$$

and consider the following partition $\sim$:

$$\mathsf{a}, \mathsf{b} \mapsto [\mathsf{a}]_\sim = \mathsf{init} \qquad \mathsf{c} \mapsto [\mathsf{c}]_\sim \qquad \mathsf{d} \mapsto [\mathsf{d}]_\sim \qquad \mathsf{e} \mapsto [\mathsf{e}]_\sim \qquad \mathsf{f}, \mathsf{g} \mapsto [\mathsf{f}]_\sim = \mathsf{good}$$

which yields the following abstract transition system:

$$\begin{tikzcd}
& {} \arrow[d] & \\
& \mathsf{init} \arrow[dl] \arrow[dr] & \\
{[\mathsf{c}]_\sim} \arrow[dr] & & {[\mathsf{d}]_\sim} \arrow[dl] \\
& {[\mathsf{e}]_\sim} \arrow[d] & \\
& \mathsf{good} \arrow[loop below] &
\end{tikzcd}$$

Assume we are interested in checking whether the property $\varphi = \mathsf{AF\ good}$ holds in the initial states of the system: to do so we need to compute $[\![\lfloor\overline{\mathsf{AF\ good}}\rfloor]\!]^\sharp_A d^\sharp$, where $d^\sharp = \alpha(\{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle\}) = \{\langle \mathsf{init}, \varnothing\rangle\}$. Using Definition 3.8 we can obtain that the mocha program associated with φ is

$$\lfloor\overline{\varphi}\rfloor = \lfloor\overline{\mathsf{AF\ good}}\rfloor = \neg\mathsf{good}?; \mathsf{push}; (\mathsf{post}; \neg\mathsf{good}?)^*; \mathsf{loop}?; \mathsf{pop}$$

The interesting part in this discussion is the computation of the abstract semantics $[\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A$ on $[\![\mathsf{push}]\!]^\sharp_A(d^\sharp) = \{\langle \mathsf{init}, \varnothing\rangle :: \langle \mathsf{init}, \varnothing\rangle\} = \{\langle \mathsf{init}, \varnothing\rangle :: S_0\}$. We have that

$$[\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A(\{\langle \mathsf{init}, \varnothing\rangle :: S_0\}) = \bigvee_{n\in\mathbb{N}} \left\{\left([\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A\right)^n \{\langle \mathsf{init}, \varnothing\rangle :: S_0\}\right\}$$

Hence (we omit the indication of the $\sim$ in the class):

- $\left([\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A\right)^0 \{\langle \mathsf{init}, \varnothing\rangle :: S_0\} = \{\langle \mathsf{init}, \varnothing\rangle :: S_0\}$
- $\left([\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A\right)^1 \{\langle \mathsf{init}, \varnothing\rangle :: S_0\} = \{\langle [\mathsf{c}], \{\mathsf{init}\}\rangle :: S_0, \langle [\mathsf{d}], \{\mathsf{init}\}\rangle :: S_0\}$
- $\left([\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A\right)^2 \{\langle [\mathsf{c}], \{\mathsf{init}\}\rangle :: S_0, \langle [\mathsf{d}], \{\mathsf{init}\}\rangle :: S_0\} = \{\langle [\mathsf{e}], \{\mathsf{init}, [\mathsf{c}], [\mathsf{d}]\}\rangle :: S_0\}$
- $\left([\![(\mathsf{post}; \neg\mathsf{good}?)^*]\!]^\sharp_A\right)^3 \{\langle [\mathsf{e}], \{\mathsf{init}, [\mathsf{c}], [\mathsf{d}]\}\rangle :: S_0\} = \cancel{\{\langle \mathsf{good}, \{\mathsf{init}, [\mathsf{c}], [\mathsf{d}], , [\mathsf{e}]\}}\rangle :: S_0\} = \varnothing$

4.3 Non partition-based abstraction

We present here a different way to lift Galois connection $(\alpha_\Sigma, \Sigma, A_\Sigma, \gamma_\Sigma)$ on states to stacks: we consider an abstraction on states that is not necessarily a partition (e.g. Intervals for the Integers domain), and we abstract it in a coarse way, in which we lose the correspondence between the states, their pasts and the stacks. For each level of the stack (inductively) we group together all the current states and all the traces, and we apply the abstraction to these two sets: thus an abstract element will be a single stack.

4.3.1 Galois Connection

As mentioned above, the abstraction will be built in two steps, and by induction on the length n of stacks of the concrete element c_n. In the following we indicate with $\sigma^\sharp, \rho^\sharp \in A_\Sigma$ elements of the abstract domain of states A_Σ, and we indicate by $\bot_\Sigma$ the bottom element of A_Σ. A *path* in the abstract domain is a pair $\langle \sigma^\sharp, \rho^\sharp \rangle$, and a stack $S^{\sharp(n+1)}$ has the form

$$S^{\sharp(n+1)} = \langle \sigma^\sharp, \rho^\sharp \rangle :: S^{\sharp n}$$

In the analysis we will see this abstraction is indeed very coarse grained, we think however it will be worth to carry out some experiments to see how it performs in practice, and how refinement can help us improve it.

Definition 4.26 (Abstract domain, abstract element)**.** The **abstract domain** is

$$\mathsf{A} = \bigcup_n \mathsf{A}^n \qquad \mathsf{A}^n = (A_\Sigma \setminus \{\bot_\Sigma\} \times A_\Sigma) \cup \{\bot\}$$

An **abstract element** is an abstract stack of some length n:

$$a_n = \langle \sigma^\sharp, \rho^\sharp \rangle :: S^{\sharp n}$$

We now define the abstraction α and the concretization γ by induction on the length n of the stack. The abstraction is build by means of two auxiliary functions. The first function u_n is meant to flatten the stack, by grouping all the states and all the traces pairwise, then the abstraction is performed on the collected stack, and acts component-wise as the function α_Σ of the Galois connection on states.

Definition 4.27 (Auxiliary functions $u_n, \overline{\alpha}_n$)**.** The map u_n is inductively defined as follows:

$$u_1 : \mathsf{C}^1 \to (2^\Sigma \times 2^\Sigma) \qquad u_n : \mathsf{C}^n \to (2^\Sigma \times 2^\Sigma) \times (2^\Sigma \times 2^\Sigma)^{(n-1)}$$

$$u_1(\{\langle \sigma_j, T_j \rangle \mid j \in J\}) = \left\langle \bigcup_{j \in J} \{\sigma_j\}, \bigcup_{j \in J} T_j \right\rangle$$

$$u_n\left(\left\{\langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I\right\}\right) = \left\langle \bigcup_{i \in I} \{\sigma_i\}, \bigcup_{i \in I} T_i \right\rangle :: u_{n-1}\left(\bigcup_{i \in I} \left\{S_i^{(n-1)}\right\}\right)$$

The map $\overline{\alpha}_n$ is inductively defined as follows:

$$\overline{\alpha}_1 : 2^\Sigma \times 2^\Sigma \to \mathsf{A}^1 \qquad \overline{\alpha}_n : (2^\Sigma \times 2^\Sigma) \times (2^\Sigma \times 2^\Sigma)^{(n-1)} \to \mathsf{A}^n$$

$$\overline{\alpha}_1\left(\left\langle \bigcup_{j\in J}\{\sigma_j\}, \bigcup_{j\in J} T_j \right\rangle\right) = \left\langle \alpha_\Sigma\left(\bigcup_{j\in J}\{\sigma_j\}\right), \alpha_\Sigma\left(\bigcup_{j\in J} T_j\right)\right\rangle$$

$$\begin{aligned}\overline{\alpha}_n\left(\left\langle \bigcup_{i\in I}\{\sigma_i\}, \bigcup_{i\in I} T_i \right\rangle :: u_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\right) &= \\ = \left\langle \alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right), \alpha_\Sigma\left(\bigcup_{i\in I} T_i\right)\right\rangle &:: \overline{\alpha}_{n-1}\circ u_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\end{aligned}$$

Definition 4.28 (Abstraction map)**.** The **abstraction map** $\alpha_n : \mathsf{C}^n \to \mathsf{A}^n$ is defined as $\alpha_n = \overline{\alpha}_n \circ u_n$.

Definition 4.29 (Concretization map)**.** The **concretization map** $\gamma_n : \mathsf{A}^n \to \mathsf{C}^n$ is inductively defined as

$$\begin{aligned}\gamma_1\left(\langle \sigma^\sharp, \rho^\sharp\rangle\right) &= \{\langle \sigma, T\rangle \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp)\} \\ \gamma_n\left(\langle \sigma^\sharp, \rho^\sharp\rangle :: S^{\sharp(n-1)}\right) &= \left\{\langle \sigma, T\rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp),\ S \in \gamma_{n-1}(S^{\sharp(n-1)})\right\}\end{aligned}$$

Proposition 4.30. The above construction $(\alpha, \mathsf{C}, \mathsf{A}, \gamma)$ is a Galois Connection.

Proof. We give the proof by induction on the length n of the stacks, by showing that each $\gamma_n \circ \alpha_n$ is a closure. First let $c = \{\langle \sigma_i, T_i\rangle :: S_i \mid i \in I\}$, then we explicitly compute

$$\begin{aligned}\alpha_n(c) &= \left\langle \alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right), \alpha_\Sigma\left(\bigcup_{i\in I} T_i\right)\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\{S_i\}\right) \\ (\gamma_n\circ\alpha_n)(c) &= \left\{\langle\sigma, T\rangle :: S \mid \sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\right),\right. \\ &\qquad \left. T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I} T_i\right)\right),\ S \in \gamma_{n-1}\left(\alpha_{n-1}\left(\bigcup_{i\in I}\{S_i\}\right)\right)\right\}\end{aligned}$$

which will be useful in the following.

($n = 1$) **(Monotone)** Let $c_1, c_2 \in \mathsf{C}^1$ be such that $c_1 \leq_{\mathsf{C}} c_2$, where

$$c_1 = \{\langle \sigma_i, T_i\rangle \mid i \in I\} \qquad c_2 = \{\langle \sigma_j, T_j\rangle \mid j \in J\}$$

and $c_1 \leq_{\mathsf{C}} c_2$ means that for all $\langle \sigma, T\rangle \in c_1$ we have $\langle \sigma, T\rangle \in c_2$. Now by definition

$$\begin{aligned}(\gamma_1\circ\alpha_1)(c_1) &= \left\{\langle\sigma, T\rangle \mid \sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\right), T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I} T_i\right)\right)\right\} \\ (\gamma_1\circ\alpha_1)(c_2) &= \left\{\langle\sigma, T\rangle \mid \sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{j\in J}\{\sigma_j\}\right)\right), T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{j\in J} T_j\right)\right)\right\}\end{aligned}$$

Then, since by hypothesis $c_1 \leq_{\mathsf{C}} c_2$, we have

$$\bigcup_{i\in I}\{\sigma_i\} \subseteq \bigcup_{j\in J}\{\sigma_j\} \qquad\qquad \bigcup_{i\in I} T_i \subseteq \bigcup_{j\in J} T_j$$

and since by hypothesis $\gamma_\Sigma \circ \alpha_\Sigma$ is a closure we have

$$\sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\right) \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{j\in J}\{\sigma_j\}\right)\right)$$
$$T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right) \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{j\in J}T_j\right)\right)$$

hence if $\langle \sigma, T\rangle \in (\gamma_1 \circ \alpha_1)(c_1)$ then $\langle \sigma, T\rangle \in (\gamma_1 \circ \alpha_1)(c_2)$.

(Idempotent) Let $c = c_1$ of the previous computation, then we can explicitly compute $(\alpha_1 \circ \gamma_1 \circ \alpha_1)(c)$ as:

$$\begin{aligned}(\alpha_1 \circ \gamma_1 \circ \alpha_1)(c) &= \left\langle \alpha_\Sigma\left(\gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\right)\right), \alpha_\Sigma\left(\gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right)\right)\right\rangle \\ &= \left\langle \alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right), \alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right\rangle = \alpha_1(c)\end{aligned}$$

since by hypothesis $\gamma_\Sigma \circ \alpha_\Sigma$ is a closure. Then since $(\alpha_1 \circ \gamma_1 \circ \alpha_1)(c) = \alpha_1(c)$, it follows that $(\gamma_1 \circ \alpha_1)^2(c) = (\gamma_1 \circ \alpha_1 \circ \gamma_1 \circ \alpha_1)(c) = (\gamma_1 \circ \alpha_1)(c)$.

(Extensive) Let $\langle \sigma, T\rangle \in c$, then $\langle \sigma, T\rangle \in (\gamma_1 \circ \alpha_1)(c)$ since $\sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\right)$ and $T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right)$ since by hypothesis $\gamma_\Sigma \circ \alpha_\Sigma$ is a closure.

$(n-1 \Rightarrow n)$ **(Monotone)** Let $c_1, c_2 \in \mathsf{C}^n$ be such that $c_1 \leq_{\mathsf{C}} c_2$, where

$$c_1 = \left\{\langle \sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i \in I\right\} \qquad c_2 = \left\{\langle \sigma_j, T_j\rangle :: S_j^{(n-1)} \mid j \in J\right\}$$

and $c_1 \leq_{\mathsf{C}} c_2$ means that for all $\langle \sigma, T\rangle :: S \in c_1$ we have $\langle \sigma, T\rangle :: S \in c_2$. Now by definition

$$\begin{aligned}(\gamma_n \circ \alpha_n)(c_1) = \Bigg\{ &\langle \sigma, T\rangle :: S \mid \sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\right), \\ &T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right),\ S \in \gamma_{n-1}\left(\alpha_{n-1}\left(\bigcup_{i\in I}\{S_i\}\right)\right)\Bigg\} \\ (\gamma_n \circ \alpha_n)(c_2) = \Bigg\{ &\langle \sigma, T\rangle :: S \mid \sigma \in \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{j\in J}\{\sigma_j\}\right)\right), \\ &T \subseteq \gamma_\Sigma\left(\alpha_\Sigma\left(\bigcup_{j\in J}T_j\right)\right),\ S \in \gamma_{n-1}\left(\alpha_{n-1}\left(\bigcup_{j\in J}\{S_j\}\right)\right)\Bigg\}\end{aligned}$$

Then, since by hypothesis $c_1 \leq_{\mathsf{C}} c_2$, we have

$$\bigcup_{i\in I}\{\sigma_i\} \subseteq \bigcup_{j\in J}\{\sigma_j\} \qquad \bigcup_{i\in I}T_i \subseteq \bigcup_{j\in J}T_j \qquad \bigcup_{i\in I}\{S_i\} \subseteq \bigcup_{j\in J}\{S_j\}$$

and since by hypothesis $\gamma_\Sigma \circ \alpha_\Sigma$ is a closure, and by inductive hypothesis $\gamma_{n-1} \circ \alpha_{n-1}$ is also a closure we have

$$\sigma \in \gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \right) \subseteq \gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{j \in J} \{\sigma_j\} \right) \right)$$

$$T \subseteq \gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right) \subseteq \gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{j \in J} T_j \right) \right)$$

$$S \in \gamma_{n-1} \left(\alpha_{n-1} \left(\bigcup_{i \in I} \{S_i\} \right) \right) \subseteq \gamma_{n-1} \left(\alpha_{n-1} \left(\bigcup_{j \in J} \{S_j\} \right) \right)$$

hence if $\langle \sigma, T \rangle : S \in (\gamma_n \circ \alpha_n)(c_1)$ then $\langle \sigma, T \rangle :: S \in (\gamma_n \circ \alpha_n)(c_2)$.

(Idempotent) Let $c = c_1$ of the previous computation, then we can explicitly compute $(\alpha_n \circ \gamma_n \circ \alpha_n)(c)$ as:

$$\begin{aligned}(\alpha_n \circ \gamma_n \circ \alpha_n)(c) &= \left\langle \alpha_\Sigma \left(\gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \right) \right), \alpha_\Sigma \left(\gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right) \right) \right\rangle \\ &\quad :: \alpha_{n-1} \left(\gamma_{n-1} \left(\alpha_{n-1} \left(\bigcup_{i \in I} \{S_i\} \right) \right) \right) \\ &= \left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \{S_i\} \right) = \alpha_n(c)\end{aligned}$$

since by hypothesis $\gamma_\Sigma \circ \alpha_\Sigma$ is a closure and by inductive hypothesis $\gamma_{n-1} \circ \alpha_{n-1}$ is also a closure. Then since $(\alpha_n \circ \gamma_n \circ \alpha_n)(c) = \alpha_n(c)$, it follows that $(\gamma_n \circ \alpha_n)^2(c) = (\gamma_n \circ \alpha_n \circ \gamma_n \circ \alpha_n)(c) = (\gamma_n \circ \alpha_n)(c)$.

(Extensive) Let $\langle \sigma, T \rangle :: S \in c$, then $\langle \sigma, T \rangle :: S \in (\gamma_n \circ \alpha_n)(c)$ since $\sigma \in \gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \right)$ and $T \subseteq \gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right)$ since by hypothesis $\gamma_\Sigma \circ \alpha_\Sigma$ is a closure, and $S \in \gamma_{n-1} \left(\alpha_{n-1} \left(\bigcup_{i \in I} \{S_i\} \right) \right)$ since by inductive hypothesis $\gamma_{n-1} \circ \alpha_{n-1}$ is also a closure. $\square$

4.3.2 Abstract Semantics

We now compute the Best Correct Approximations for basic expressions and study under which conditions on concrete elements we have local completeness. We will see that for the push and pop operators we actually have global completeness.

$\llbracket \mathsf{p}? \rrbracket^{\sharp n}_{A}$ computation Consider $a_n = \langle \sigma^\sharp, T^\sharp \rangle :: S^{\sharp(n-1)}$. Then we have:

$$\begin{aligned}
\llbracket \mathsf{p}? \rrbracket^{\sharp n}_{A}(a) &= (\alpha_n \llbracket \mathsf{p}? \rrbracket_n \gamma_n)(a) = (\overline{\alpha}_n u_n \llbracket \mathsf{p}? \rrbracket_n \gamma_n) \left(\langle \sigma^\sharp, T^\sharp \rangle :: S^{\sharp(n-1)} \right) = \\
&= (\overline{\alpha}_n u_n \llbracket \mathsf{p}? \rrbracket_n) \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp),\ S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_n u_n \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp), \sigma \models \mathsf{p},\ T \subseteq \gamma_\Sigma(\rho^\sharp),\ S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_n u_n \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp) \cap \mathsf{p},\ T \subseteq \gamma_\Sigma(\rho^\sharp),\ S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_n \left(\langle \gamma_\Sigma(\sigma^\sharp) \cap \mathsf{p}, \gamma_\Sigma(\rho^\sharp) \rangle :: u_{n-1} \left(\gamma_{n-1} \left(S^{\sharp(n-1)} \right) \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma(\sigma^\sharp) \cap \mathsf{p} \right), \alpha_\Sigma \gamma_\Sigma(\rho^\sharp) \right\rangle :: \overline{\alpha}_{n-1} u_{n-1} \left(\gamma_{n-1} \left(S^{\sharp(n-1)} \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma(\sigma^\sharp) \cap \mathsf{p} \right), \alpha_\Sigma \gamma_\Sigma(\rho^\sharp) \right\rangle :: \alpha_{n-1} \gamma_{n-1} \left(S^{\sharp(n-1)} \right)
\end{aligned}$$

In order to have local completeness, we must check under which conditions $\alpha_n \circ \llbracket \mathsf{p}? \rrbracket_n = \llbracket \mathsf{p}? \rrbracket^{\sharp n}_{A} \circ \alpha_n$. Consider $c_n = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\}$, then

$$\begin{aligned}
(\alpha_n \circ \llbracket \mathsf{p}? \rrbracket_n)\, c_n &= \alpha_n \left(\{ \langle \sigma_i, T_i \rangle :: S_i \mid \sigma_i \models \mathsf{p},\ i \in I \} \right) \\
&= \overline{\alpha}_n \circ u_n \left(\{ \langle \sigma_i, T_i \rangle :: S_i \mid \sigma_i \models \mathsf{p},\ i \in I \} \right) \\
&= \overline{\alpha}_n \left(\left\langle \left(\bigcup_{i \in I} \{\sigma_i\} \right) \cap \mathsf{p}, \bigcup_{i \in I, \sigma_i \models \mathsf{p}} T_i \right\rangle :: u_{n-1} \left(\bigcup_{i \in I, \sigma_i \models \mathsf{p}} \left\{ S_i^{(n-1)} \right\} \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\left(\bigcup_{i \in I} \{\sigma_i\} \right) \cap \mathsf{p} \right), \alpha_\Sigma \left(\bigcup_{i \in I, \sigma_i \models \mathsf{p}} T_i \right) \right\rangle :: \overline{\alpha}_{n-1} \circ u_{n-1} \left(\bigcup_{i \in I, \sigma_i \models \mathsf{p}} \left\{ S_i^{(n-1)} \right\} \right) \\
(\llbracket \mathsf{p}? \rrbracket^{\sharp n}_{A} \circ \alpha_n) c_n &= \llbracket \mathsf{p}? \rrbracket^{\sharp n}_{A} \left(\left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \overline{\alpha}_{n-1} \circ u_{n-1} \left(\bigcup_{i \in I} \{ S_i^{n-1} \} \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \right) \cap \mathsf{p} \right), \alpha_\Sigma \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \gamma_{n-1} \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \right) \cap \mathsf{p} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right)
\end{aligned}$$

Hence the issues to address to achieve local completeness are:

1. $\alpha_\Sigma \left(\gamma_\Sigma \left(\alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \right) \cap \mathsf{p} \right) \overset{?}{=} \alpha_\Sigma \left(\left(\bigcup_{i \in I} \{\sigma_i\} \right) \cap \mathsf{p} \right)$

2. $\alpha_\Sigma \left(\bigcup_{i \in I, \sigma_i \models \mathsf{p}} T_i \right) \overset{?}{=} \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right)$

3. $\alpha_{n-1} \left(\bigcup_{i \in I, \sigma_i \models \mathsf{p}} \left\{ S_i^{(n-1)} \right\} \right) = \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right)$

where 1. holds if p is expressible in the domain A_Σ, since $\alpha\gamma\alpha = \alpha$, but in general 2. and 3. are not true: the problem is that the abstraction with the union eliminates every relation between the current state and its trace.

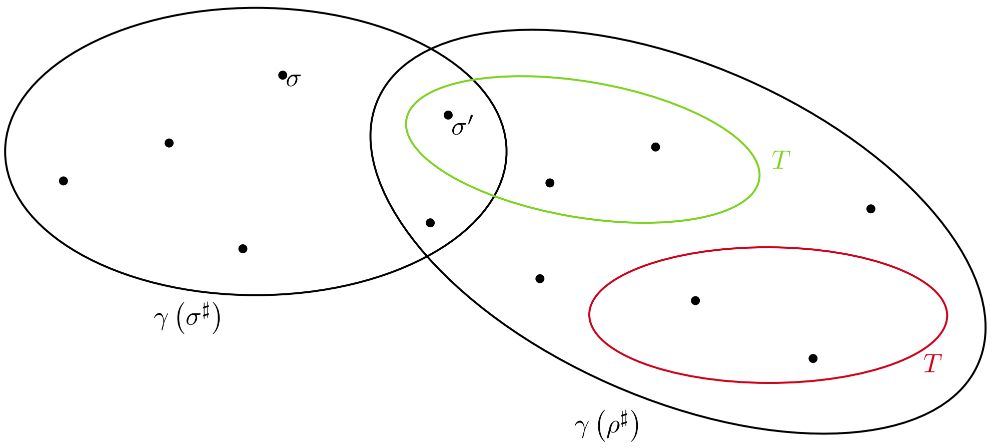
σ
σ'
T
T
$\gamma\left(\sigma^\sharp\right)$
$\gamma\left(\rho^\sharp\right)$

Figure 4.6: Visual representation for the $[\![\mathsf{loop?}]\!]^\sharp_A$.

$[\![\mathsf{loop?}]\!]^{\sharp n}_A$ computation Consider $a_n = \left\langle \sigma^\sharp, T^\sharp \right\rangle :: S^{\sharp(n-1)}$. Then we have:

$$\begin{aligned}
[\![\mathsf{loop?}]\!]^{\sharp n}_A(a) &= (\alpha_n [\![\mathsf{loop?}]\!]_n \gamma_n)(a) = (\overline{\alpha}_n u_n [\![\mathsf{p?}_n]\!] \gamma_n) \left(\left\langle \sigma^\sharp, T^\sharp \right\rangle :: S^{\sharp(n-1)} \right) = \\
&= (\overline{\alpha}_n u_n [\![\mathsf{loop?}]\!]_n) \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp),\ S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_n u_n \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ \sigma \in T,\ T \subseteq \gamma_\Sigma(\rho^\sharp),\ S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_n \left(\langle \gamma_\Sigma(\sigma^\sharp) \cap \gamma_\Sigma(\rho^\sharp), \gamma_\Sigma(\rho^\sharp) \rangle \right) :: \alpha_{n-1}\gamma_{n-1} \left(S^{\sharp(n-1)} \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma(\sigma^\sharp) \cap \gamma_\Sigma(\rho^\sharp) \right), \alpha_\Sigma \gamma_\Sigma(\rho^\sharp) \right\rangle :: \alpha_{n-1}\gamma_{n-1} \left(S^{\sharp(n-1)} \right)
\end{aligned}$$

Figure 4.6 illustrates the intuition behind the fact that the resulting abstract state is $\alpha_\Sigma \left(\gamma_\Sigma(\sigma^\sharp) \cap \gamma_\Sigma(\rho^\sharp) \right)$.

In order to have local completeness, we must check under which conditions $\alpha_n \circ [\![\mathsf{loop?}]\!]_n = [\![\mathsf{loop?}]\!]^{\sharp n}_A \circ \alpha_n$. Consider $c_n = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\}$, then

$$\begin{aligned}
(\alpha_n \circ [\![\mathsf{loop?}]\!]_n)\, c_n &= \alpha_n \left(\{ \langle \sigma_i, T_i \rangle :: S_i \mid \sigma_i \in T_i,\ i \in I \} \right) \\
&= \overline{\alpha}_n \circ u_n \left(\{ \langle \sigma_i, T_i \rangle :: S_i \mid \sigma_i \in T_i,\ i \in I \} \right) \\
&= \overline{\alpha}_n \left(\left\langle \bigcup_{i \in I, \sigma_i \in T_i} \{\sigma_i\}, \bigcup_{i \in I, \sigma_i \in T_i} T_i \right\rangle :: u_{n-1} \left(\bigcup_{i \in I, \sigma_i \in T_i} \left\{ S_i^{(n-1)} \right\} \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\bigcup_{i \in I, \sigma_i \in T_i} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I, \sigma_i \in T_i} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I, \sigma_i \in T_i} \left\{ S_i^{(n-1)} \right\} \right) \\
([\![\mathsf{loop?}]\!]^{\sharp n}_A \circ \alpha_n) c_n &= [\![\mathsf{loop?}]\!]^{\sharp n}_A \left(\left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \overline{\alpha}_{n-1} \circ u_{n-1} \left(\bigcup_{i \in I} \{ S_i^{n-1} \} \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \cap \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right), \alpha_\Sigma \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle \\
&\qquad :: \alpha_{n-1}\gamma_{n-1}\alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \\
&= \left\langle \alpha_\Sigma \left(\gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right) \cap \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right)
\end{aligned}$$

hence the issues to address to achieve local completeness are

1. $\alpha_\Sigma\left(\gamma_\Sigma\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\cap\gamma_\Sigma\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right)\overset{?}{=}\alpha_\Sigma\left(\bigcup_{i\in I,\sigma_i\in T_i}\{\sigma_i\}\right)$

2. $\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\overset{?}{=}\alpha_\Sigma\left(\bigcup_{i\in I,\sigma_i\in T_i}T_i\right)$

3. $\alpha_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\overset{?}{=}\alpha_{n-1}\left(\bigcup_{i\in I,\sigma_i\in T_i}\left\{S_i^{(n-1)}\right\}\right)$

where in general 2. and 3. are not true. The condition 1. makes explicit the fact that already by performing the union we could add a loop that is not there in the concrete, and this gets worsened by using the abstraction on top of that.

$[\![\mathsf{next}]\!]^{\sharp n}_A$ **computation** Consider $a_n=\langle\sigma^\sharp,T^\sharp\rangle :: S^{\sharp(n-1)}$. Then we have:

$$\begin{aligned}
[\![\mathsf{next}]\!]^{\sharp n}_A(a) &= (\alpha_n[\![\mathsf{next}]\!]_n\gamma_n)(a)=(\overline{\alpha}_n u_n[\![\mathsf{next}]\!]_n\gamma_n)\left(\langle\sigma^\sharp,T^\sharp\rangle :: S^{\sharp(n-1)}\right)=\\
&=(\overline{\alpha}_n u_n[\![\mathsf{next}]\!]_n)\left(\left\{\langle\sigma,T\rangle :: S \mid \sigma\in\gamma_\Sigma(\sigma^\sharp),\ T\subseteq\gamma_\Sigma(\rho^\sharp),\ S\in\gamma_{n-1}(S^{\sharp(n-1)})\right\}\right)\\
&=\overline{\alpha}_n u_n\left(\left\{\langle\sigma',\varnothing\rangle :: S \mid \sigma\in\gamma_\Sigma(\sigma^\sharp),\ \sigma\twoheadrightarrow\sigma',\ S\in\gamma_{n-1}(S^{\sharp(n-1)})\right\}\right)\\
&=\overline{\alpha}_1\left(\left\langle\bigcup_{\substack{\sigma\in\gamma_\Sigma(\sigma^\sharp)\\ \sigma\twoheadrightarrow\sigma'}}\{\sigma'\},\varnothing\right\rangle\right) :: \alpha_{n-1}\gamma_{n-1}\left(S^{\sharp(n-1)}\right)\\
&=\left\langle\alpha_\Sigma\left(\bigcup_{\substack{\sigma\in\gamma_\Sigma(\sigma^\sharp)\\ \sigma\twoheadrightarrow\sigma'}}\{\sigma'\}\right),\bot\right\rangle :: \alpha_{n-1}\gamma_{n-1}\left(S^{\sharp(n-1)}\right)
\end{aligned}$$

In order to have local completeness, we must check under which conditions $\alpha_n\circ[\![\mathsf{next}]\!]_n=[\![\mathsf{next}]\!]^{\sharp n}_A\circ\alpha_n$. Consider $c_n=\left\{\langle\sigma_i,T_i\rangle :: S_i^{(n-1)} \mid i\in I\right\}$, then

$$\begin{aligned}
(\alpha_n\circ[\![\mathsf{next}]\!]_n)\,c_n &= \alpha_n\left(\{\langle\sigma_i',\varnothing\rangle :: S_i \mid \sigma_i\twoheadrightarrow\sigma_i',\ i\in I\}\right)\\
&=\overline{\alpha}_n\circ u_n\left(\{\langle\sigma_i',\varnothing\rangle :: S_i \mid \sigma_i\twoheadrightarrow\sigma_i',\ i\in I\}\right)\\
&=\overline{\alpha}_n\left(\left\langle\bigcup_{\substack{i\in I\\ \sigma_i\twoheadrightarrow\sigma_i'}}\{\sigma_i'\},\varnothing\right\rangle :: u_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\right)\\
&=\left\langle\alpha_\Sigma\left(\bigcup_{\substack{i\in I\\ \sigma_i\twoheadrightarrow\sigma_i'}}\{\sigma_i'\}\right),\bot\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\\
([\![\mathsf{next}]\!]^{\sharp n}_A\circ\alpha_n)c_n &= [\![\mathsf{next}]\!]^{\sharp n}_A\left(\left\langle\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right),\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\{S_i^{n-1}\}\right)\right)\\
&=\left\langle\alpha_\Sigma\left(\bigcup_{\substack{\sigma\in\gamma_\Sigma\alpha_\Sigma(\bigcup_{i\in I}\{\sigma_i\})\\ \sigma\twoheadrightarrow\sigma'}}\{\sigma'\}\right),\bot\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\{S_i^{n-1}\}\right)
\end{aligned}$$

Hence the issue to address to achieve local completeness is just

$$\alpha_\Sigma\left(\bigcup_{\substack{i\in I\\ \sigma_i \twoheadrightarrow \sigma_i'}} \{\sigma_i'\}\right) \overset{?}{=} \alpha_\Sigma\left(\bigcup_{\substack{\sigma\in\gamma_\Sigma\alpha_\Sigma(\bigcup_{i\in I}\{\sigma_i\})\\ \sigma\twoheadrightarrow\sigma'}} \{\sigma'\}\right)$$

which in this case relies heavily on the effect of the abstraction α_Σ other than that of the unification: in the left-hand side we only abstract on the proper successors of the states of the concrete, while on the right-hand side we abstract on the successors of all the elements that belong to the closure of the union of the concrete states.

$[\![\mathsf{post}]\!]_A^{\sharp n}$ computation Consider $a_n = \langle\sigma^\sharp, T^\sharp\rangle :: S^{\sharp(n-1)}$. Then we have:

$$\begin{aligned}
[\![\mathsf{post}]\!]_A^{\sharp n}(a) &= (\alpha_n[\![\mathsf{post}]\!]_n\gamma_n)(a) = (\overline{\alpha}_n u_n[\![\mathsf{post}]\!]_n\gamma_n)\left(\langle\sigma^\sharp, T^\sharp\rangle :: S^{\sharp(n-1)}\right) = \\
&= (\overline{\alpha}_n u_n[\![\mathsf{post}]\!]_n)\left(\left\{\langle\sigma, T\rangle :: S \mid \sigma\in\gamma_\Sigma(\sigma^\sharp),\ T\subseteq\gamma_\Sigma(\rho^\sharp),\ S\in\gamma_{n-1}(S^{\sharp(n-1)})\right\}\right)\\
&= \overline{\alpha}_n u_n\left(\left\{\langle\sigma', T\cup\{\sigma\}\rangle :: S \mid \sigma\in\gamma_\Sigma(\sigma^\sharp),\ \sigma\twoheadrightarrow\sigma',\ T\subseteq\gamma_\Sigma(\rho^\sharp),\ S\in\gamma_{n-1}(S^{\sharp(n-1)})\right\}\right)\\
&= \overline{\alpha}_1\left(\left\langle \bigcup_{\sigma\in\gamma_\Sigma(\sigma^\sharp),\sigma\twoheadrightarrow\sigma'}\{\sigma'\}, \gamma_\Sigma(\rho^\sharp)\cup\gamma_\Sigma(\sigma^\sharp)\right\rangle\right) :: \alpha_{n-1}\gamma_{n-1}\left(S^{\sharp(n-1)}\right)\\
&= \left\langle \alpha_\Sigma\left(\bigcup_{\sigma\in\gamma_\Sigma(\sigma^\sharp),\sigma\twoheadrightarrow\sigma'}\{\sigma'\}\right), \alpha_\Sigma\left(\gamma_\Sigma(\sigma^\sharp)\cup\gamma_\Sigma(\rho^\sharp)\right)\right\rangle :: \alpha_{n-1}\gamma_{n-1}\left(S^{\sharp(n-1)}\right)
\end{aligned}$$

In order to have local completeness, we must check under which conditions $\alpha_n\circ[\![\mathsf{post}]\!]_n = [\![\mathsf{post}]\!]_A^{\sharp n}\circ\alpha_n$. Consider $c_n = \left\{\langle\sigma_i, T_i\rangle :: S_i^{(n-1)} \mid i\in I\right\}$, then

$$\begin{aligned}
(\alpha_n\circ[\![\mathsf{post}]\!]_n)\, c_n &= \alpha_n\left(\{\langle\sigma_i', T_i\cup\{\sigma_i\}\rangle :: S_i \mid \sigma_i\twoheadrightarrow\sigma_i',\ i\in I\}\right)\\
&= \overline{\alpha}_n\circ u_n\left(\{\langle\sigma_i', T_i\cup\{\sigma_i\}\rangle :: S_i \mid \sigma_i\twoheadrightarrow\sigma_i',\ i\in I\}\right)\\
&= \overline{\alpha}_n\left(\left\langle \bigcup_{i\in I,\sigma_i\twoheadrightarrow\sigma_i'}\{\sigma_i'\}, \bigcup_{i\in I} T_i\cup\{\sigma_i\}\right\rangle :: u_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\right)\\
&= \left\langle \alpha_\Sigma\left(\bigcup_{i\in I,\sigma_i\twoheadrightarrow\sigma_i'}\{\sigma_i'\}\right), \alpha_\Sigma\left(\bigcup_{i\in I} T_i\cup\{\sigma_i\}\right)\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\left\{S_i^{(n-1)}\right\}\right)\\
([\![\mathsf{post}]\!]_A^{\sharp n}\circ\alpha_n)c_n &= [\![\mathsf{post}]\!]_A^{\sharp n}\left(\left\langle\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right), \alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\{S_i^{n-1}\}\right)\right)\\
&= \left\langle\alpha_\Sigma\left(\bigcup_{\substack{\sigma\in\gamma_\Sigma\alpha_\Sigma(\bigcup_{i\in I}\{\sigma_i\})\\ \sigma\twoheadrightarrow\sigma'}}\{\sigma'\}\right), \alpha_\Sigma\left(\gamma_\Sigma\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\cup\gamma_\Sigma\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right)\right\rangle :: \alpha_{n-1}\left(\bigcup_{i\in I}\{S_i^{n-1}\}\right)
\end{aligned}$$

Hence the issues to address to achieve local completeness are

1. $\alpha_\Sigma\left(\bigcup_{\substack{i\in I\\ \sigma_i\twoheadrightarrow\sigma_i'}}\{\sigma_i'\}\right) \overset{?}{=} \alpha_\Sigma\left(\bigcup_{\substack{\sigma\in\gamma_\Sigma\alpha_\Sigma(\bigcup_{i\in I}\{\sigma_i\})\\ \sigma\twoheadrightarrow\sigma'}}\{\sigma'\}\right)$
2. $\alpha_\Sigma\left(\bigcup_{i\in I}T_i\cup\{\sigma_i\}\right) \overset{?}{=} \alpha_\Sigma\left(\gamma_\Sigma\alpha_\Sigma\left(\bigcup_{i\in I}\{\sigma_i\}\right)\cup\gamma_\Sigma\alpha_\Sigma\left(\bigcup_{i\in I}T_i\right)\right)$

We can see that the condition 1. on the state is the same as that of $[\![\mathsf{next}]\!]_A^\sharp$, and that condition 2., that refers to traces, depends more on the closure of the abstraction on states than on the unification.

$[\![\mathsf{push}]\!]^\sharp_A$ **computation** Consider $a_n = \langle \sigma^\sharp, T^\sharp \rangle :: S^{\sharp(n-1)}$. Then we have:

$$\begin{aligned}
[\![\mathsf{push}]\!]^\sharp_A(a_n) &= (\alpha_n [\![\mathsf{push}]\!]_n \gamma_n)(a_n) = (\overline{\alpha}_n u_n [\![\mathsf{push}]\!] \gamma) \left(\langle \sigma^\sharp, T^\sharp \rangle :: S^{\sharp(n-1)} \right) = \\
&= (\overline{\alpha}_{n+1} u_{n+1} [\![\mathsf{push}]\!]) \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp), S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_{n+1} u_{n+1} \left(\left\{ \langle \sigma, \varnothing \rangle :: \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp), S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_2 \left(\langle \gamma_\Sigma(\sigma^\sharp), \varnothing \rangle :: \langle \gamma_\Sigma(\sigma^\sharp), \gamma_\Sigma(\rho^\sharp) \rangle \right) :: \alpha_{n-1}\gamma_{n-1}(S^{\sharp(n-1)}) \\
&= \langle \alpha_\Sigma \gamma_\Sigma(\sigma^\sharp), \bot_\Sigma \rangle :: \langle \alpha_\Sigma\gamma_\Sigma(\sigma^\sharp), \alpha_\Sigma\gamma_\Sigma(\rho^\sharp) \rangle :: \alpha_{n-1}\gamma_{n-1}(S^{\sharp(n-1)})
\end{aligned}$$

Then global completeness always holds, since $\alpha_{n+1} \circ [\![\mathsf{push}]\!]_n = [\![\mathsf{push}]\!]^{\sharp n}_A \circ \alpha_n$ always holds. Consider $c_n = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\}$, then

$$\begin{aligned}
(\alpha_{n+1} \circ [\![\mathsf{push}]\!]_n) c_n &= \overline{\alpha}_{n+1} u_{n+1} \left(\{ \langle \sigma_i, \varnothing \rangle :: \langle \sigma_i, T_i \rangle \} :: S_i \mid i \in I \right) \\
&= \overline{\alpha}_{n+1} \left(\left\langle \bigcup_{i \in I} \{\sigma_i\}, \varnothing \right\rangle :: \left\langle \bigcup_{i \in I} \{\sigma_i\}, \bigcup_{i \in I} T_i \right\rangle :: u_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \right) \\
&= \left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \bot_\Sigma \right\rangle :: \left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \\
([\![\mathsf{push}]\!]^{\sharp n}_A \circ \alpha_n) c_n &= [\![\mathsf{push}]\!]^{\sharp n}_A \left(\left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \right) \\
&= \left\langle \alpha_\Sigma \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \bot_\Sigma \right\rangle :: \left\langle \alpha_\Sigma \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \gamma_\Sigma \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle \\
&\qquad :: \alpha_{n-1} \gamma_{n-1} \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \\
&= \left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \bot_\Sigma \right\rangle :: \left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right)
\end{aligned}$$

$[\![\mathsf{pop}]\!]^\sharp_A$ **computation** Consider $a_n = \langle \sigma^\sharp, T^\sharp \rangle :: S^{\sharp(n-1)}$. Then we have:

$$\begin{aligned}
[\![\mathsf{pop}]\!]^\sharp_A(a_n) &= (\alpha_{n-1} [\![\mathsf{pop}]\!] \gamma)(a_n) = (\overline{\alpha}_{n-1} u_{n-1} [\![\mathsf{pop}]\!] \gamma) \left(\langle \sigma^\sharp, T^\sharp \rangle :: S^{\sharp(n-1)} \right) = \\
&= (\overline{\alpha}_{n-1} u_{n-1} [\![\mathsf{pop}]\!]) \left(\left\{ \langle \sigma, T \rangle :: S \mid \sigma \in \gamma_\Sigma(\sigma^\sharp),\ T \subseteq \gamma_\Sigma(\rho^\sharp), S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) \\
&= \overline{\alpha}_{n-1} u_{n-1} \left(\left\{ S \mid S \in \gamma_{n-1}(S^{\sharp(n-1)}) \right\} \right) = \overline{\alpha}_{n-1} u_{n-1} \gamma_{n-1}(S^\sharp) \\
&= \alpha_{n-1} \gamma_{n-1}(S^\sharp)
\end{aligned}$$

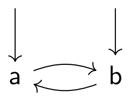

Figure 4.7: A simple example using the non-partition based abstraction (Ex. 4.31).

Then global completeness always holds, since $\alpha_{n-1} \circ [\![\mathsf{pop}]\!]_n = [\![\mathsf{pop}]\!]_A^{\sharp n} \circ \alpha_n$ always holds. Consider $c_n = \left\{ \langle \sigma_i, T_i \rangle :: S_i^{(n-1)} \mid i \in I \right\}$, then

$$\begin{aligned}
(\alpha_{n-1} \circ [\![\mathsf{pop}]\!]_n) c_n &= \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \\
([\![\mathsf{pop}]\!]_A^{\sharp n} \circ \alpha_n) c_n &= [\![\mathsf{pop}]\!]_A^{\sharp n} \left(\left\langle \alpha_\Sigma \left(\bigcup_{i \in I} \{\sigma_i\} \right), \alpha_\Sigma \left(\bigcup_{i \in I} T_i \right) \right\rangle :: \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \right) \\
&= \alpha_{n-1} \gamma_{n-1} \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right) \\
&= \alpha_{n-1} \left(\bigcup_{i \in I} \left\{ S_i^{(n-1)} \right\} \right)
\end{aligned}$$

We introduce now a simple example in which $\alpha_\Sigma = \mathrm{id}$ to show that this kind of abstraction heavily loses information, even when the underlying abstraction on states is the identity.

Example 4.31. Consider the transition system of Figure 4.7 with $\alpha_\Sigma = \mathrm{id}$. We can see that the abstract domain can successfully prove that the property $\varphi = \mathsf{AF\ a}$ holds: we show the relevant parts of the proof derivation for the property in Figure 4.8 (where we have set $S = \langle \mathsf{b}, \varnothing \rangle$). Using Definition 3.8 we can obtain that the mocha program associated with φ is

$$\begin{aligned}
\lfloor \overline{\varphi} \rfloor = \lfloor \overline{\mathsf{AF\ a}} \rfloor &= \lfloor \bar{\mathsf{a}} \rfloor ; \mathsf{push}; (\mathsf{post}; \lfloor \bar{\mathsf{a}} \rfloor)^* ; \mathsf{loop}?; \mathsf{pop} \\
&= \neg \mathsf{a}?; \mathsf{push}; (\mathsf{post}; \neg \mathsf{a}?)^*; \mathsf{loop}?; \mathsf{pop}
\end{aligned}$$

We can see that the four proof obligations $\mathbb{C}^A_{\{\langle \mathsf{a}, \varnothing \rangle, \langle \mathsf{b}, \varnothing \rangle\}}(\neg \mathsf{a}?)$, $\mathbb{C}^A_{\varnothing}(\mathsf{loop}?)$, $\mathbb{C}^A_{\{\langle \mathsf{b}, \varnothing \rangle :: S\}}(\mathsf{post})$ and $\mathbb{C}^A_{\{\langle \mathsf{a}, \{\mathsf{b}\} \rangle :: S\}}(\mathsf{a}?)$ are satisfied because the domain satisfies the conditions studied for the local completeness (in all of these cases, the tail of the stack poses no issue): the conditions for a? are easily checked since a is indeed expressible in $A_\Sigma = \Sigma$ and the condition on loop? is vacuously true on the empty set. The condition on post is to be read as

$$\mathbb{C}^A_{\{\langle \mathsf{b}, \varnothing \rangle :: S\}}(\mathsf{post}) \qquad \Leftrightarrow \qquad (A \circ [\![\mathsf{post}]\!] \circ A)(\{\langle \mathsf{b}, \varnothing \rangle :: S\}) = (A \circ [\![\mathsf{post}]\!])(\{\langle \mathsf{b}, \varnothing \rangle :: S\})$$

which holds, since we have $A(\{\langle \mathsf{b}, \varnothing \rangle :: S\}) = \{\langle \mathsf{b}, \varnothing \rangle :: S\}$, i.e. this set is representable in the abstract domain.

We notice that, however, this simple abstraction is not globally complete. Let $c = \{\langle \mathsf{a}, \varnothing \rangle, \langle \mathsf{b}, \varnothing \rangle\}$ be the set of initial states and consider for example $c' = [\![\mathsf{post}]\!](c) = \{\langle \mathsf{a}, \{\mathsf{b}\} \rangle, \langle \mathsf{b}, \{\mathsf{a}\} \rangle\}$. Then we have

$$\dfrac{\dfrac{\mathbb{C}^A_{\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle\}}(\neg\mathsf{a}?)}{\vdash_A [\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle\}]\ \neg\mathsf{a}?\ [\{\langle \mathsf{b},\varnothing\rangle\}]}\,(\text{trsf}) \qquad \dfrac{\vdots}{\vdash_A [\{\langle \mathsf{b},\varnothing\rangle\}]\ (\mathsf{push};(\mathsf{post};\neg\mathsf{a}?)^*;\mathsf{loop}?;\mathsf{pop})\ [\varnothing]}\,(\text{seq})}{\vdash_A [\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle\}]\ (\neg\mathsf{a}?;\mathsf{push};(\mathsf{post};\neg\mathsf{a}?)^*;\mathsf{loop}?;\mathsf{pop})\ [\varnothing]}\,(\text{seq})$$

$$\dfrac{\dfrac{\vdots}{\vdash_A [\{\langle \mathsf{b},\varnothing\rangle :: S\}]\ (\mathsf{post};\mathsf{a}?)^*\ [\varnothing]}\,(\text{it}) \qquad \dfrac{\mathbb{C}^A_{\varnothing}(\mathsf{loop}?)}{\vdash_A [\varnothing]\ \mathsf{loop}?\ [\varnothing]}\,(\text{trsf})}{\vdash_A [\{\langle \mathsf{b},\varnothing\rangle :: S\}]\ (\mathsf{post};\mathsf{a}?)^*;\mathsf{loop}?\ [\varnothing]}\,(\text{seq})$$

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^A_{\{\langle \mathsf{b},\varnothing\rangle :: S\}}(\mathsf{post})}{\vdash_A [\{\langle \mathsf{b},\varnothing\rangle :: S\}]\ \mathsf{post}\ [\{\langle \mathsf{a},\{\mathsf{b}\}\rangle :: S\}]}\,(\text{trsf}) \qquad \dfrac{\mathbb{C}^A_{\{\langle \mathsf{a},\{\mathsf{b}\}\rangle :: S\}}(\mathsf{a}?)}{\vdash_A [\{\langle \mathsf{a},\{\mathsf{b}\}\rangle :: S\}]\ \mathsf{a}?\ [\varnothing]}\,(\text{trsf})}{\vdash_A [\{\langle \mathsf{b},\varnothing\rangle :: S\}]\ (\mathsf{post};\mathsf{a}?)\ [\varnothing] \qquad \varnothing \subseteq A(\{\langle \mathsf{b},\varnothing\rangle :: S\})}\,(\text{seq})}{\vdash_A [\{\langle \mathsf{b},\varnothing\rangle :: S\}]\ (\mathsf{post};\mathsf{a}?)^*\ [\varnothing]}\,(\text{it})$$

Figure 4.8: Parts of the proof tree for the property $\varphi = \mathsf{AF}\ \mathsf{a}$ for the simple non-partition based example (Ex. 4.31).

$\alpha(c') = \langle\{\mathsf{a},\mathsf{b}\},\{\mathsf{a},\mathsf{b}\}\rangle$ and we can see that the $\mathsf{loop}?$ command is not complete on c':

$$(\llbracket\mathsf{loop}?\rrbracket^\sharp_A \circ \alpha)(c') = \llbracket\mathsf{loop}?\rrbracket^\sharp_A(\langle\{\mathsf{a},\mathsf{b}\},\{\mathsf{a},\mathsf{b}\}\rangle) = \langle\{\mathsf{a},\mathsf{b}\},\{\mathsf{a},\mathsf{b}\}\rangle$$
$$(\alpha \circ \llbracket\mathsf{loop}?\rrbracket)(c') = \alpha(\varnothing) = \bot$$

We reprise an example discussed before to compare the result of this abstraction with the one introduced in Section 4.2, on which we will later show a small example of repair (Ex. 4.38).

Example 4.32. Consider again the transition system discussed in Examples 3.10 and 4.22:

$$\begin{tikzcd} {} \arrow[d] & {} \arrow[d] \\ \mathsf{a} \arrow[d] & \mathsf{b} \arrow[l] \\ \mathsf{c} \arrow[loop below] & \end{tikzcd}$$

and consider the following abstraction α_Σ, that let us distinguish b,c but identifies a with b:

$$\alpha_\Sigma(\{\mathsf{b}\}) = \mathbf{b} \qquad \alpha_\Sigma(\{\mathsf{a},\mathsf{b}\}) = \mathbf{ab} \qquad \alpha_\Sigma(\{\mathsf{c}\}) = \mathbf{c}$$

with $\alpha_\Sigma(\varnothing) = \bot$, $\alpha_\Sigma(\{\mathsf{a},\mathsf{c}\}) = \{\mathsf{b},\mathsf{c}\} = \{\mathsf{a},\mathsf{b},\mathsf{c}\} = \top$. Hence the Hasse diagram of the abstract domain lattice is represented in Figure 4.9 This abstraction introduces the same spurious loop of $\{\mathsf{a},\mathsf{b}\}$ on itself, which introduces a spurious error alarm in the computation of $\varphi' = \mathsf{AF}\ \mathsf{c}$. The proof obligation that fails in the derivation tree is the same as Example 4.22:

$$\dfrac{\mathbb{C}^A_{\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle,\langle \mathsf{a},\{\mathsf{b}\}\rangle\}}(\mathsf{loop}?)}{\vdash_A [\{\langle \mathsf{a},\varnothing\rangle,\langle \mathsf{b},\varnothing\rangle,\langle \mathsf{a},\{\mathsf{b}\}\rangle\}]\ \mathsf{loop}?\ [\varnothing]}\,(\text{transfer})$$

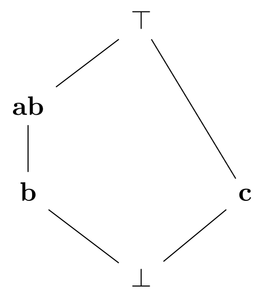

Figure 4.9: Hasse diagram for the non-partitioning abstraction α_Σ (Ex. 4.32).

and we can compute

$$\begin{aligned}
[\![\mathsf{loop?}]\!](\{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle, \langle \mathsf{a}, \{\mathsf{b}\}\rangle\}) &= \varnothing \\
[\![\mathsf{loop?}]\!]\gamma\alpha(\{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle, \langle \mathsf{a}, \{\mathsf{b}\}\rangle\}) &= [\![\mathsf{loop?}]\!]\gamma(\langle \mathbf{ab}, \mathbf{b}\rangle) \\
&= [\![\mathsf{loop?}]\!]\,\{\langle \mathsf{a}, \varnothing\rangle, \langle \mathsf{b}, \varnothing\rangle, \langle \mathsf{a}, \{\mathsf{b}\}\rangle, \langle \mathsf{b}, \{\mathsf{b}\}\rangle\} \neq \varnothing
\end{aligned}$$

4.4 An outlook on Abstract Interpetation Repair

One of the problems addressed in [29] is, roughly speaking, that of deriving the most abstract domain which allows one to decide program correctness without raising false alarms. A key notion is that of a *complete abstract domain*, which, however, is intrinsically a *global* notion: i.e. it aims at ensuring the absence of false alarms for all possible inputs. A complete abstract domain for a given transfer function f can be obtained by constructing the complete shell with respect to f, which makes the abstract domain complete for f on all possible inputs. This typically results in a domain which is very close to the concrete domain. The idea introduced with Abstract Interpretation Repair is to focus on a *single execution* produced by the transfer function on some input of interest.

Repair. Whenever the current abstract domain is not precise enough to prevent false-alarms, AIR can be used to *optimally* refine the domain locally to the inputs of interest to remove these false-alarms. The general scenario involves a composite transfer function $f = f_n \circ \cdots \circ f_1$ which models the sequential composition of $f_1, \ldots, f_n : C \to C$, a concrete input c and a correctness specification a for which the aim is to decide if $f(c) \leq a$ or not. In Abstract Interpretation we select an abstract domain A such that a is *expressible* in A and then we perform the check of the validity in the abstract domain, namely we check that $f^\sharp_A \alpha(c) \leq_A a$ holds, where $f^\sharp_A = f^A_n \circ \cdots \circ f^A_1$ and each f^A_i is the *best correct approximation* of f_i in A, i.e. $f^A_i = \alpha \circ f_i \circ \gamma$.

If the specification is verified in the abstract domain we can conclude by soundness that it is also verified in the concrete domain; on the other hand if it is not verified, if $f^\sharp \circ \alpha(c) = \alpha \circ f(c)$, then from $f^\sharp_A \alpha(c) \not\leq_A a$ we can conclude that the specification is not verified in the concrete domain.

Definition 4.33 (Domain Refinements)**.** [29, Sec. 3] Given $A \in \mathrm{Abs}(C)$ and $N \subseteq C$, we define the **least refinement** of A including the new elements in N as

$$A \boxplus N = \mathcal{M}(\gamma_A(A) \cup N)$$

where $\mathcal{M}$ denotes the Moore closure (Def. 2.5)

Remark 4.34 (Pointed refinement)**.** A special case happens when refinement happens adding only one new element, i.e. $N = \{z\}$, and in this case we write $A_z = A \boxplus \{z\}$ and we have $A_z(c) = z \wedge A(c)$ if $c \leq z$, $A_z(c) = c$ otherwise. The A_z is the **pointed refinement** of A.

That of pointed refinement is a core notion, since we want to repair to achieve completeness by adding the minimum number of new abstract points. The minimal refinement would be the exact shell, that may not exists ([29, Ex. 4.6]), so we turn our focus on pointed refinements. In particular [29, Lemma 4.7] illustrates which pointed refinements may achieve local completeness. Then the authors introduce the novel definition of *pointed* shell refinement on abstract domains.

Definition 4.35 (Pointed Shells)**.** [29, Def. 4.8] Let $f : C \to C$ be monotone, $A \in \mathrm{Abs}(C)$ and $c \in C$. The **pointed locally complete shell** (*pointed shell* for short) of A in c exists when

$$\max\left(\left\{x \in C \mid x \leq A(c),\ \mathbb{C}_c^{A_x}(f)\right\}\right) = \{u\}$$

and, in such a case, the pointed shell is $A_u \in \mathrm{Abs}(C)$.

Then [29, Th. 4.9] shows how to construct the pointed shell, when it exists.

Definition 4.36 (Local completeness set)**.** [29, Def. 4.3] Let $f : C \to C$, $A \in \mathrm{Abs}(C)$ and $c \in C$. The **local completeness set** $\mathbb{L}_{c,f}^A \subseteq C$ is defined as

$$\mathbb{L}_{c,f}^A = \{x \in C \mid x \leq A(c),\ f(x) \leq Af(c)\}$$

Theorem 4.37. [29, Th. 4.9] Let $f : C \to C$, $A \in \mathrm{Abs}(C)$, $c \in C$ and let $u = \bigvee \mathbb{L}_{c,f}^A$. If f is monotone then $\mathbb{C}_c^{A_u}(f)$ if and only if A_u is the pointed shell of A for f on c, and if f is additive then $\mathbb{C}_c^{A_u}(f)$ if and only if $(f(c) \leq u \Rightarrow f(u) \leq u)$.

The theorem implies that for an additive function f (such as, for example, the collecting semantics), and a new concrete element $u = \bigvee \mathbb{L}_{c,f}^A$, the pointed shell of A exists when either $f(c) \not\leq u$ or $f(u) \leq u$, and in that case A_u is precisely the pointed shell.

Forward Repair. The forward repair strategy works by processing the local completeness proof obligations regarding each f_i that combined ensure the completeness of the composite transfer function f. Let $c_k = (f_{k-1} \circ \dots f_1)(c)$ and $c_1 = c$, then the equality $f_A^\sharp A(c) = A(f(c))$ comes as consequence of the n local completeness proof obligations

$$\forall k \in [1, n] \qquad (A \circ f_k)(c_k) = (A \circ f_k \circ A)(c_k)$$

These proof obligations are processed in increasing order, and as soon as the smallest index k such that the proof obligation is violated is found, the domain A gets repaired with the pointed shell A_u and the analysis starts again on the new domain. The element added is $u = \vee \mathbb{L}^{A}_{c_k, f_k}$, exploiting Theorem 4.37, when it exists, else c_k is added.

Forward repair stops when the obtained domain is precise enough to prove the correctness of the specification or such that all the local completeness requirements are met. The worst case happens when the specification is not correct, because all local completeness equations must be processed and all c_k must be computed.

A comment on backward repair. Forward repair is not the only possible approach: in the article the authors introduce also a backward repair strategy, which focuses on weakest liberal preconditions (instead of strongest postconditions c_k, as the forward procedure does). The two strategies compute the pointed shells for different concrete elements, resulting in general in different refined domains. While the forward repair strategy is easier to formulate, since it is evident its link with the underlying Local Completeness Logic, the backward repair has some prominent strengths: it mainly operates on the abstract domain and not on concrete elements, and successive repairs can be don along the existing abstract computation (so there is no need to redo the abstract interpretation, as in the case of the forward repair).

Example 4.38. Consider the abstraction introduced in Example 4.32, and consider the proof obligation that fails because $\mathbb{C}^{A}_{\{\langle \mathsf{a}, \varnothing \rangle, \langle \mathsf{b}, \varnothing \rangle, \langle \mathsf{a}, \{\mathsf{b}\} \rangle\}}(\mathsf{loop?})$ does not hold. If we were to perform the repair on this domain, the element we need to add to A^1 is, letting $c' = \{\langle \mathsf{a}, \varnothing \rangle, \langle \mathsf{b}, \varnothing \rangle, \langle \mathsf{a}, \{\mathsf{b}\} \rangle\}$:

$$
\begin{aligned}
u = \bigvee \mathbb{L}^{A}_{c, \llbracket \mathsf{loop?} \rrbracket} &= \bigvee \left\{ x \in \mathsf{C}^1 \mid x \leq \gamma\alpha(c'),\ \mathsf{loop?}(x) \leq \gamma\alpha\llbracket \mathsf{loop?} \rrbracket(c') \right\} \\
&= \bigcup X \subseteq \Sigma \times 2^{\Sigma} \mid X \subseteq \{\langle \mathsf{a}, \varnothing \rangle, \langle \mathsf{b}, \varnothing \rangle, \langle \mathsf{a}, \{\mathsf{b}\} \rangle, \langle \mathsf{b}, \{\mathsf{b}\} \rangle\}, \llbracket \mathsf{loop?} \rrbracket(X) = \varnothing \\
&= \{\langle \mathsf{a}, \varnothing \rangle, \langle \mathsf{b}, \varnothing \rangle, \langle \mathsf{a}, \{\mathsf{b}\} \rangle\}
\end{aligned}
$$

Notice that this refinement happens on the *stack domain*, not on the underlying domain of states.

Conclusion

We devised a framework that lets us encode the problem of Model Checking ACTL formulae on finite transition systems as programs in a novel core language, called mocha (Model Checking as Abstract Interpretation). The encoding of an ACTL formula φ is realised in a way that the semantics of the program $\lfloor\overline{\varphi}\rfloor$ computed on a state σ is the empty set if and only if the property holds on that state. If this is not the case, i.e. the property does not hold, the state is returned as counterexample. More generally, when $\lfloor\overline{\varphi}\rfloor$ is executed on a set of states V it returns exactly the subset of states $\{\sigma \in V \mid \sigma \not\models \varphi\}$ for which φ does not hold. The language actually operates on a domain of stacks of finite length, to allow for recursive verification of nested formulae and to maintain an abstracted version of the traversed path in the form of set of traversed states.

This encoding allows one to view Model Checking as an instance of program verification in order to allow for the reuse of the vast theory and toolset of Abstract Interpretation. The thesis focuses on ACTL, the universal fragment of the temporal logic CTL, which admits existential counterexamples.

The idea of encoding the Model Checking problem in a program verification task was inspired by the novel approach to an algorithmic way of refining abstract domains to achieve local, instead of global, completeness introduced in [29]. Hence, after building a concrete setup for the computation of counterexamples different abstractions are introduced which are to be intended in the framework of Local Completeness Logic [32].

We discuss two possible ways of lifting abstractions of the states to abstractions on the stacks. The first technique applies to partition-based abstractions (equivalences on the set of states). This is in line to what is done in the CEGAR approach. The second abstraction applies to general abstractions on the power set of states.

We also consider a "basic" abstraction which can be actually shown to be complete for the fragment of the mocha language used for encoding ACTL. Thanks to a simple but effective result about the possibility of combining complete and locally complete abstractions preserving local completeness, this can be possibly used in combination with partition abstraction.

Development directions. The work in this thesis opens the way to a number of future developments both on the theoretical and practical side. The thesis focuses on ACTL, the fragment of CTL* where only universal quantification on paths is admitted and temporal operators are always paired with quantifications. A very natural direction of further research thus consists in trying to deal with wider fragments of the logic. The first natural candidate would be ACTL*, where the second restriction mentioned above is lifted, i.e. (universal) quantifications and temporal operators can be used independently. The study of ACTL* could open the way of framing CEGAR as an instance of this technique. Moreover, the semantics could be enriched to generate counterexamples that are linear traces or trees for the selected states, i.e. to not only find the states in which the property does not hold, but also to extract the reason why it fails. As it happens in the CEGAR approach this could be useful in designing the repair (see below). Note that this would properly generalise CEGAR, which can only deal with linear counterexamples.

One could consider the implementation of the technique, taking into account problems related to the efficient representation of abstract domains and operations. This could allow experimenting with the different ways of lifting the abstraction and including the usage of the repair by Abstract Interpretation.

Bibliography

[1] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[2] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[3] Patrick Cousot and Radhia Cousot. “Systematic Design of Program Analysis Frameworks”. In: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Barry K. Rosen. ACM Press, 1979, pp. 269–282. DOI: 10.1145/567752.567778.

[4] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. “Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach”. In: *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983*. Ed. by John R. Wright et al. ACM Press, 1983, pp. 117–126. DOI: 10.1145/567067.567080.

[5] Randal E. Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Trans. Computers* 35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[6] Kim Guldstrand Larsen and Bent Thomsen. “A Modal Process Logic”. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS ’88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pp. 203–210. DOI: 10.1109/LICS.1988.5119.

[7] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model Checking and Abstraction”. In: *ACM Trans. Program. Lang. Syst.* 16.5 (1994), pp. 1512–1542. DOI: 10.1145/186025.186051.

[8] Claire Loiseaux et al. “Property Preserving Abstractions for the Verification of Concurrent Systems”. In: *Formal Methods Syst. Des.* 6.1 (1995), pp. 11–44. DOI: 10.1007/BF01384313.

[9] Dennis Dams, Rob Gerth, and Orna Grumberg. “Abstract Interpretation of Reactive Systems”. In: *ACM Trans. Program. Lang. Syst.* 19.2 (1997), pp. 253–291. DOI: 10.1145/244795.244800.

[10] Dexter Kozen. “Kleene Algebra with Tests”. In: *ACM Trans. Program. Lang. Syst.* 19.3 (1997), pp. 427–443. DOI: 10.1145/256167.256195.

[11] Gérard Le Lann. “An analysis of the Ariane 5 flight 501 failure-a system engineering perspective”. In: *1997 Workshop on Engineering of Computer-Based Systems (ECBS ’97), March 24-28, 1997, Monterey, CA, USA*. IEEE Computer Society, 1997, pp. 339–246. DOI: 10.1109/ECBS.1997.581900.

[12] Ph. Lacan et al. “ARIANE 5 - The Software Reliability Verification Process”. In: *DASIA 98 - Data Systems in Aerospace*. Ed. by B. Kaldeich-Schürmann. Vol. 422. ESA Special Publication. July 1998, p. 201.

[13] Patrick Cousot and Radhia Cousot. “Temporal Abstract Interpretation”. In: *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. Ed. by Mark N. Wegman and Thomas W. Reps. ACM, 2000, pp. 12–25. DOI: 10.1145/325694.325699.

[14] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. “Making abstract interpretations complete”. In: *J. ACM* 47.2 (2000), pp. 361–416. DOI: 10.1145/333979.333989.

[15] Francesco Buccafurri et al. “On ACTL Formulas Having Linear Counterexamples”. In: *J. Comput. Syst. Sci.* 62.3 (2001), pp. 463–515. DOI: 10.1006/jcss.2000.1734.

[16] David Schmidt. “Binary Relations for Abstraction and Refinement”. In: (Jan. 2001).

[17] Armin Biere et al. “Bounded model checking”. In: *Adv. Comput.* 58 (2003), pp. 117–148. DOI: 10.1016/S0065-2458(03)58003-2.

[18] Edmund M. Clarke et al. “Counterexample-guided abstraction refinement for symbolic model checking”. In: *J. ACM* 50.5 (2003), pp. 752–794. DOI: 10.1145/876638.876643.

[19] Orna Grumberg et al. “When not losing is better than winning: Abstraction and refinement for the full mu-calculus”. In: *Inf. Comput.* 205.8 (2007), pp. 1130–1148. DOI: 10.1016/j.ic.2006.10.009.

[20] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[21] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. “Analyzing Program Analyses”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 261–273. DOI: 10.1145/2676726.2676987.

[22] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *CoRR* abs/1801.01203 (2018). arXiv: 1801.01203.

[23] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 973–990.

[24] Antoine Miné. *Static Inference of Numeric Invariants by Abstract Interpretation*. Course Notes, Université Pierre et Marie Curie, Paris. 2018.

[25] Patrick Cousot. "Calculational Design of a Regular Model Checker by Abstract Interpretation". In: *Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings*. Ed. by Robert M. Hierons and Mohamed Mosbah. Vol. 11884. Lecture Notes in Computer Science. Springer, 2019, pp. 3–21. DOI: 10.1007/978-3-030-32505-3_1.

[26] Igor Konnov. "Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (eds): Handbook of model checking - Springer International Publishing AG, Cham, Switzerland, 2018". In: *Formal Aspects Comput.* 31.4 (2019), pp. 455–456. DOI: 10.1007/s00165-019-00486-z.

[27] Paolo Baldan, Barbara König, and Tommaso Padoan. "Abstraction, Up-To Techniques and Games for Systems of Fixpoint Equations". In: *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*. Ed. by Igor Konnov and Laura Kovács. Vol. 171. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 25:1–25:20. DOI: 10.4230/LIPIcs.CONCUR.2020.25.

[28] Peter W. O'Hearn. "Incorrectness logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 10:1–10:32. DOI: 10.1145/3371078.

[29] Roberto Bruni et al. "Abstract interpretation repair". In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 426–441. DOI: 10.1145/3519939.3523453.

[30] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. "Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498721.

[31] Roberto Giacobazzi and Francesco Ranzato. "History of Abstract Interpretation". In: *IEEE Ann. Hist. Comput.* 44.2 (2022), pp. 33–43. DOI: 10.1109/MAHC.2021.3133136.

[32] Roberto Bruni et al. "A Correctness and Incorrectness Program Logic". In: *J. ACM* 70.2 (2023), 15:1–15:45. DOI: 10.1145/3582267.