UNIVERSITÀ DI PISA

DEPARTMENT OF COMPUTER SCIENCE

Ph.D. Thesis

Combining over and under-approximation for program analysis

Candidate
Flavio Ascari

Advisors
Roberto Bruni
Roberta Gori

Academic year 2024/2025

Abstract

Formal static analysis has always been mainly focused on over-approximation to prove correctness. Recently, the dual approach of under-approximation has gained attention as a formal basis to prove incorrectness. While the two paradigms can be applied separately, it has been shown that their are able to cooperate to decide more effectively both correctness and incorrectness. In this thesis, we study analogies and differences between over and under-approximation to understand what can and cannot be ported from the well studied over-approximation theory to the less studied under-approximation one, and try to combine the two approaches to improve both. We first focus on abstract interpretation, finding limitations in approaches based on under-approximating abstract domains. We then turn our attention to program logics, classifying known results as over or under-approximation of forward or backward semantics, and defining a new backward-oriented proof system for backward under-approximation. Then we present novel extensions of two approaches combining over and under-approximation in non-trivial ways, namely Local Completeness Logic and Property Directed Reachability, showing how they improve on proving both correctness and incorrectness of the system under analysis.

Contents

Chapter 1

Introduction

Rewrite from scratch: expand with motivation and results

Static program analyses are techniques used to infer properties of programs directly from their source code, without executing them. They have been studied and successfully applied for over 50 years to produce effective methods and tools to support the development of correct software. For all these years, the main focus of static analysis was to prove *correctness* property of software such as, for instance, the absence of bugs or a security requirement. To this end, static analyses compute *over-approximations*, ie. supersets of all possible behaviours, of the semantics of programs: the absence of unwanted behaviour in the over-approximation guarantees the correctness of the program. However, over-approximation cannot be used to disprove correctness, eg. by exposing real bugs, since any alert raised by the analyser may be caused by the over-approximation rather than by the program, ie. it can be a so called false alarm.

Hoare logic [Hoa69] is perhaps the first example of formal static analysis, and indeed is an over-approximation one, that correctly fits its goal of proving the absence of errors. Maybe early works like this and influential opinions such as Dijkstra's renown quote "*Program testing can be used to show the presence of bugs, but never to show their absence!*" [Dij70] directed the focus towards over-approximation. However, from the point of view of a software developer, false alarms are undesirable because they undermine the credibility and usefulness of the analysis. Following this observation, O'Hearn recently argued for the relevance of formal methods for bug catching, and more in general for disproving correctness – in other words, proving *incorrectness*. With this in mind, he proposed Incorrectness Logic [OHe20], a dual version of Hoare logic thought from the ground up for this goal. Incorrectness logic takes static analysis upside down, as it computes an *under-approximation* of the semantics, ie. a subset of all possible behaviours of a program. Dually to over-approximation, under-approximation can then expose defects in the code, but it is unable to show their absence. Given the recent introduction of this new idea, we believe the field has many interesting topics yet to study.

When these two paradigms are taken separately, it is possible to either prove or disprove correctness. However, their combination proved to be more effective. The insight is that even partial information about correctness can help the search for an incorrectness proof and vice versa. The goal of the proposed research is then to study the effect of this interaction, especially with respect to abstract interpretation-based techniques. Abstract interpretation [CC77; RY20] is a general framework based on constructive approximations to define, compare and combine sound static analysis. Given its flexibility, it has been applied to many fields of computer science, such as program transformation, semantics and security [CC14]. Because of this, we believe it can turn out as an effective tool also in this new setting of combining over- and under-approximations, which hasn't been fully explored yet.

This proposal is organized as follows. Chapter 2 lays the background needed to understand subsequent chapters. Chapter 3 examines other works that combines over- and under-approximations: they are useful to draw a comparison with our initial results and take inspiration for future enhancements.

Chapter 2

Background

In this chapter, we lay the background for the thesis. We fix the notation and recall known notions on different aspects of theoretical computer science.

2.1 Order structures

We write $\mathcal{P}(S)$ for the powerset of S and $\mathrm{id}_S : S \to S$ for the identity function on a set S. We omit subscripts when obvious from the context. If $f : S \to T$ is a function, we overload the symbol f to denote also its lifting $f : \mathcal{P}(S) \to \mathcal{P}(T)$ defined as $f(X) = \{f(x) \mid x \in X\}$ for any $X \subseteq S$. Given two functions $f : S \to T$ and $g : T \to V$ we denote their composition as $g \circ f$ or simply gf. For a function $f : S \to S$, we denote $f^n : S \to S$ the composition of f with itself n times, i.e., $f^0 = \mathrm{id}_S$ and $f^{n+1} = f \circ f^n$.

In order structures (such as posets and lattices) with carrier set C, we denote the ordering with $\leq_C$, least upper bounds (lubs) with $\vee_C$, greatest lower bounds (glbs) with $\wedge_C$, least element with $\bot_C$ and greatest element with $\top_C$. For all these, we omit the subscript when clear from the context. We recall that any powerset is a complete lattice with ordering given by inclusion. In this case, we use standard symbols $\subseteq$, $\cup$, etc. If C is a poset, we denote by C^{op} the poset with the same carrier set but reverse ordering: $a \leq_{C^{\mathrm{op}}} b$ if and only if $b \leq_C a$. Given a poset T and two functions $f, g : S \to T$, the notation $f \leq g$ means that, for all $s \in S$, $f(s) \leq_T g(s)$. A function f between complete lattices is additive (resp. co-additive) whenever it preserves arbitrary lubs (resp. glbs). Given a function $f : C \to C$ on a poset C, we call a point $x \in C$ a fixed point (or fixpoint) if $f(x) = x$, and write both $\mathrm{lfp}(f)$ and $\mu(f)$ to denote its least fixpoint, if it exists. We recall two standard results guaranteeing the existence of (least) fixpoints (see, e.g., [DP02]):

Theorem 2.1 (Knaster-Tarski). *Let L be a complete lattice and let $f : L \to L$ be a monotone function. Then the set of fixed points of f is a complete lattice.*

Corollary 2.2. *Since a complete lattice cannot be empty, lfp(f) exists.*

Theorem 2.3 (Kleene). *Let C be a complete partial order and $f : C \to C$ a Scott-continuous function. Then lfp(f) is the least upper bound of the chain*

$$\bot \leq f(\bot) \leq f^2(\bot) \leq f^3(\bot) \leq \dots$$

The above chain is also called the *initial chain* of f. If instead of C we consider C^{op}, by duality we get that whenever f is Scott-co-continuous the greatest fixpoint of f, denoted by $\mathrm{gfp}(f)$, is the greatest lower bound of the *final chain* of f

$$\dots \leq f^3(\top) \leq f^2(\top) \leq f(\top) \leq \top.$$

If we assume f is additive, then for any point $x \in C$ we have $(f \vee x)^n = \bigvee_{j<n} f^j(x)$. By Kleene, this means $\mathrm{lfp}(f \vee x) = \bigvee_{n \geq 0} f^n(x)$. Dually, if f is co-additive, $\mathrm{gfp}(f \wedge x) = \bigwedge_{n \geq 0} f^n(x)$.

Suppose $f \dashv g$. Then, by Knaster-Tarski,

$$\mathrm{lfp}(f \vee x) \leq y \iff x \leq \mathrm{gfp}(g \wedge y). \tag{2.1}$$

Lastly, from Knaster-Tarski follows the following two (co)induction proof principles:

$$\frac{\exists x. f(x) \leq x \leq p}{\mathrm{lfp}(f) \leq p} \qquad\qquad \frac{\exists x. p \leq x \leq f(x)}{p \leq \mathrm{gfp}(f)} \tag{2.2}$$

2.2 Propositional logic

When talking about propositional logical formulas, we group variables x_1, x_2, …, x_n in a vector denoted $\overline{x}$. Moreover, we often omit the variables a formula depends on. For instance, if the formula $F(\overline{x})$ depends on variables $\overline{x}$, we shall write just F. We use the special shorthand F' to denote $F[\overline{x}/\overline{x}']$, where we primed all variables appearing in F. An assignment s of all variables $\overline{x}$ appearing in a formula $F(\overline{x})$ either satisfies it, written $s \vDash F$, or falsifies it, written $s \nvDash F$. Given an assignment s for variables $\overline{x}$, we shall write s' for the corresponding assignment on primed variables $\overline{x}'$, that is the value of variable x' in s' is the same as x in s. Moreover, we use the comma to "merge" assignments when they refer to disjoint sets of variables: for instance, given the two assignments s and t, we write s, t' for the assignment that uses s for $\overline{x}$ and t' for $\overline{x}'$ (we assume the set of primed variables $\overline{x}'$ to be disjoint from the original set $\overline{x}$). A *literal* is either a variable or its negation. A *clause* c is a disjunction of literals. Any assignment s of variables $\overline{x}$ has a corresponding clause that is satisfied exactly by that assignment of the same variables, and we overload the symbol s to denote both. A *subclause* $d \subseteq c$ is a clause whose literals are a subset of literals of c. A formula F in conjunctive normal form (CNF) is a conjunction of clauses, and we write $\mathrm{clause}(F)$ to denote the set of clauses that appear in F.

stuff for first-
logic?

2.3 Regular commands

A common setting to study static analysis is a simple while-language, which contains basic constructs needed to write imperative programs [Win93]. However, we consider a different system that is able to encode the standard while-language, namely we focus on regular commands (as done in many recent works [OHe20; BGGR21; MOH21; Raa+20; ZDS23]):

$$\mathsf{Reg} \ni \mathsf{r} ::= \ \mathsf{e} \mid \mathsf{r};\mathsf{r} \mid \mathsf{r} \oplus \mathsf{r} \mid \mathsf{r}^{\star} \tag{2.3}$$

Regular commands can be instantiated differently by changing the set Exp of basic transfer expressions e. These determines the kind of basic operations allowed in the language. For instance, when Exp contains deterministic assignments and boolean guards it can encode a standard while-language (discussed below). Another example is Kleene algebras with tests [Koz97] (see Section 2.8).

The command $\mathsf{r};\mathsf{r}$ represent the usual sequential composition. $\mathsf{r} \oplus \mathsf{r}$ is nondeterministic choice. The Kleene star $\mathsf{r}^{\star}$ denote a nondeterministic iteration, where r can be executed any number of time (possibly 0) before exiting. It can be thought as the solution of the recursive equation $\mathsf{r}^{\star} \equiv \texttt{skip} \oplus (\mathsf{r};\mathsf{r}^{\star})$. We write r^n to denote sequential composition of r with itself n times, analogously to how we use f^n for function composition.

$$
\begin{aligned}
[\![\mathsf{e}]\!] &\triangleq (\!|\mathsf{e}|\!) c\\
[\![\mathsf{r}_1; \mathsf{r}_2]\!] c &\triangleq [\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!](c)\\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!] c &\triangleq [\![\mathsf{r}_1]\!] c \vee [\![\mathsf{r}_2]\!] c\\
[\![\mathsf{r}^\star]\!] c &\triangleq \bigvee_{n \geq 0} [\![\mathsf{r}]\!]^n c
\end{aligned}
$$

Figure 2.1: Generic semantics of regular commands.

For the generic semantics of regular commands, we assume a concrete set of values C that is a complete join-semilattice. We also assume the semantics $(\!|\cdot|\!) : \mathsf{Exp} \to C \to C$ to be given – in general, it depends on the instantiation. The semantics of regular commands $[\![\cdot]\!] : \mathsf{Reg} \to C \to C$ is defined inductively in Figure 2.1. Intuitively, this defines the collecting semantics of a program: if C is the powerset of the set of states, $[\![\mathsf{r}]\!] c$ is the set of output states reachable from the set of input states c.

To recover the standard while-language, using standard definitions for arithmetic and boolean expressions $\mathsf{a} \in \mathsf{AExp}$ and $\mathsf{b} \in \mathsf{BExp}$, we let

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

The command `skip` does nothing. `x := a` is a standard deterministic assignment. `b?` is an "assume" statement: it filters out inputs that falsify `b`. With this set of expressions, regular commands are very similar to Dijkstra's guarded commands [Dij75]. We can define `if` and `while` statements as syntactic sugar:

$$
\begin{aligned}
\texttt{if (b) then c}_1\texttt{ else c}_2 &\triangleq (\texttt{b?; c}_1) \oplus ((\neg\texttt{b})\texttt{?; c}_2)\\
\texttt{while (b) do c} &\triangleq (\texttt{b?; c})^\star ; (\neg\texttt{b})\texttt{?}
\end{aligned}
$$

For the semantics of these basic expressions, we consider a finite set of variables Var, then the set of stores $\Sigma \triangleq \mathrm{Var} \to \mathbb{Z}$ that are (total) functions σ from Var to integers. The complete lattice C is defined as $\mathcal{P}(\Sigma)$ with the usual order structure given by set inclusion. Given a store $\sigma \in \Sigma$, $\sigma[x \mapsto v]$ denotes function update as usual for $x \in \mathrm{Var}$ and $v \in \mathbb{Z}$. We consider standard, inductively defined semantics $(\!|\cdot|\!)$ for arithmetic and boolean expressions $\mathsf{a} \in \mathrm{AExp}$ and $\mathsf{b} \in \mathrm{BExp}$. The semantics of expressions $\mathsf{e} \in \mathsf{Exp}$ is then defined as

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

This defines the collecting denotational semantics of programs. In fact, it is easy to check that the semantics of `if` and `while` is the standard one for while-language. We also remark that this semantics $(\!|\cdot|\!)$ is both monotone and additive, and these properties are lifted to the semantics of regular commands $[\![\cdot]\!]$:

Proposition 2.4. *If $(\!|\cdot|\!)$ is monotone (resp. additive), then the semantics $[\![\cdot]\!]$ defined as in Figure 2.1 is monotone (resp. additive) as well.*

As notation, we will write $[\![\mathsf{r}]\!]\sigma$ instead of $[\![\mathsf{r}]\!]\{\sigma\}$.

$$\frac{}{\{P\}\ \mathsf{e}\ \{[\![\mathsf{e}]\!]P\}}\ \{\mathsf{atom}\} \qquad \frac{P \Rightarrow P' \quad \{P'\}\ \mathsf{r}\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ \mathsf{r}\ \{Q\}}\ \{\mathsf{cons}\}$$

$$\frac{\{P\}\ \mathsf{r}_1\ \{R\} \quad \{R\}\ \mathsf{r}_2\ \{Q\}}{\{P\}\ \mathsf{r}_1; \mathsf{r}_2\ \{Q\}}\ \{\mathsf{seq}\} \qquad \frac{\{P\}\ \mathsf{r}_1\ \{Q\} \quad \{P\}\ \mathsf{r}_2\ \{Q\}}{\{P\}\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \{Q\}}\ \{\mathsf{choice}\}$$

$$\frac{\{P\}\ \mathsf{r}\ \{P\}}{\{P\}\ \mathsf{r}^\star\ \{P\}}\ \{\mathsf{iter}\}$$

Figure 2.2: Hoare logic for regular commands

2.4 Hoare logic

Hoare logic (HL for short) [Hoa69] is a triple-based logic that proves properties about programs. Given two assertions P, Q and a regular command r,[1] the HL triple

$$\{P\}\ \mathsf{r}\ \{Q\}$$

means that, whenever the execution of r begins in a state σ satisfying P and it ends in a state σ', then σ' satisfies Q. When Q is a correctness specification, any HL triple $\{P\}\ \mathsf{r}\ \{Q\}$ provides a *sufficient* condition P for the so called partial correctness of the program r. Formally, given the semantics $[\![\cdot]\!]$ of regular commands and abusing notation by writing P instead of the set of states satisfying that formula, the validity of an HL triple is given by over-approximation property of postconditions

$$[\![\mathsf{r}]\!]P \subseteq Q. \tag{HL}$$

An HL triple $\{P\}\ \mathsf{r}\ \{Q\}$ is *valid*, written $\models \{P\}\ \mathsf{r}\ \{Q\}$, if the condition (HL) holds.

In its original formulation, HL was proposed for a deterministic while-language and assuming P and Q were first-order logic formulae. Subsequent work generalized it to many other settings, such as nondeterminism [Apt84] and regular commands [MOH21], resulting in the rules in Figure 2.2. That is a minimal set of correct rules for HL, but we remark that there are many other valid rules other than those in the figure. We also point out that rule $\{\mathsf{iter}\}$ is based on *invariants*, properties whose validity is preserved by the loop body. From this, if an invariant is true at the beginning, it is true also after any number of iterations. Invariants are a simple, yet sound and complete proof technique for loop over-approximation. We use the standard notation $\vdash \{P\}\ \mathsf{r}\ \{Q\}$ to express that a triple is *provable* in the HL proof system.

Theorem 2.5 (HL is sound [HL74])**.** *All provable triples in HL are valid:*

$$\vdash \{P\}\ \mathsf{r}\ \{Q\} \implies \models \{P\}\ \mathsf{r}\ \{Q\}$$

The reverse implication is called *completeness* of the logic,[2] and in general it does not hold for HL because program analysis is undecidable. Particularly, when we use first-order logic formulae as assertions, HL is not complete because first-order logic is not able to

[1] The original formulation uses the while-language. However, since regular commands are the primary syntax used in this thesis, we focus on them instead.

[2] For the sake of readability, we must warn the reader that this thesis deals with three different notions of completeness. (Logical) completeness, used here, refers to the ability of a proof system to derive all valid triples. Global and local completeness are instead properties of abstract interpretation, that will be defined later. We will make sure to disambiguate the term whenever it is not clear from the context.

$$\frac{}{[P]\ \mathsf{e}\ [\llbracket \mathsf{e} \rrbracket P]}\ \textsf{[atom]} \qquad \frac{P \supseteq P' \quad [P']\ \mathsf{r}\ [Q'] \quad Q' \supseteq Q}{[P]\ \mathsf{r}\ [Q]}\ \textsf{[cons]}$$

$$\frac{[P]\ \mathsf{r}_1\ [R] \quad [R]\ \mathsf{r}_2\ [Q]}{[P]\ \mathsf{r}_1; \mathsf{r}_2\ [Q]}\ \textsf{[seq]} \qquad \frac{[P]\ \mathsf{r}_1\ [Q_1] \quad [P]\ \mathsf{r}_2\ [Q_2]}{[P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \cup Q_2]}\ \textsf{[choice]}$$

$$\frac{\forall n \geq 0 \,.\, [P_n]\ \mathsf{r}\ [P_{n+1}]}{[P_0]\ \mathsf{r}^\star\ [\bigcup_{n \geq 0} P_n]}\ \textsf{[iter]}$$

Figure 2.3: Simplified Incorrectness Logic, adapted from [MOH21].

represent all the properties needed to prove completeness, notably loop invariants [Apt81, §2.7]. Cook [Coo78] overcame these limitations for the first time by adding an oracle to decide implications and requiring that all strongest post (including loop-invariants) are expressible in the assertion language, proving the first completeness result for HL.[3] Another approach (see, e.g., [CCLB12; OHe20; Zil24]) is to assume P and Q to be *set* of states instead of formulae in some assertion language. This allows us to dodge at once both the issue of expressibility of the assertion language (since we can write any subset) and of decidability of the implication (since it reduces to subset inclusion). This latter approach is sometimes called ”semantics assertions”, as opposed to the ”syntactic assertions” approach of using formulae in some language. In this thesis, we will consider semantics assertions where P and Q are sets of states, with the only exception of Separation SIL (Section 5.4). Therefore, we will write $\sigma \in P$ to mean that state σ satisfies the assertion P, and use standard set-theoretic symbols such as $\subseteq$ and $\cup$ instead of the logical $\implies$ and $\vee$.

When P and Q are sets of states, the HL proof system in Figure 2.2 is complete:

Theorem 2.6 (HL is complete [MOH21])**.** *All valid triples in HL are provable:*

$$\models \{P\}\ \mathsf{r}\ \{Q\} \implies \vdash \{P\}\ \mathsf{r}\ \{Q\}$$

Lastly, we remark that HL is tightly connected to Dijkstra's weakest liberal precondition [Dij75]. Given a program r and a predicate Q on final states, the weakest liberal precondition $\mathbf{wlp}[\mathsf{r}](Q)$ is a predicate on initial states such that a state σ satisfies $\mathbf{wlp}[\mathsf{r}](Q)$ if and only if the execution of r starting from σ either doesn't terminate or terminates in a state satisfying Q. This definition is reminiscent of the validity condition for HL triples, but the "if and only if" requirement makes it stronger: $\mathbf{wlp}[\mathsf{r}](Q)$ is the *weakest* precondition such that $\models \{\mathbf{wlp}[\mathsf{r}](Q)\}\ \mathsf{r}\ \{Q\}$. In other words, $\models \{P\}\ \mathsf{r}\ \{Q\}$ if and only if $P \implies \mathbf{wlp}[\mathsf{r}](Q)$. While Dijkstra's work focused on giving an inductive definition for $\mathbf{wlp}$, we focus on it as a sort of inverse of $\llbracket\cdot\rrbracket$.[4] As discussed above, we also consider $\mathbf{wlp}\mathsf{r}(Q)$ to be a set of states instead of a formula.

2.5 Incorrectness Logic

Incorrectness Logic (IL) [OHe20] was introduced as a formalism for under-approximation with the idea of finding true bugs in the code. The IL triple

$$[P]\ \mathsf{r}\ [Q]$$

[3] This was later called completeness *in the sense of Cook* [Apt81, §2.8].

[4] More precisely, $\mathbf{wlp}$ and $\llbracket\cdot\rrbracket$ are adjoint functions, see Section 5.3.3.

means that all the states in Q are reachable from states in P. Therefore, any error state in Q is a true error of the program and the analysis can report it safely to developers. Formally, validity of an IL triple is given by the following formula

$$\forall \sigma' \in Q \,.\, \exists \sigma \in P \,.\, \sigma' \in [\![\mathsf{r}]\!]\sigma \tag{IL$_{\text{FOL}}$}$$

which can be compactly rewritten as the under-approximation condition

$$[\![\mathsf{r}]\!]P \supseteq Q \tag{IL}$$

This property ensures that any error in Q reported by the analysis is in fact a true error of the program, reachable in some concrete execution.

We consider the proof system for IL in Figure 2.3, adapting the one in [MOH21] and inspired by previous work on reverse Hoare logic [VK11], simplified to not separate correct and erroneous termination states. Rule [cons] looks similar, but uses reversed implication: in IL we can grow the precondition ($P' \subseteq P$) and shrink the post ($Q \subseteq Q'$), while in HL we do the opposite (cf. $\{\mathsf{cons}\}$ in Figure 2.2). This fits the under/over-approximation duality: if we know Q' is an under-approximation of the result, also $Q \subseteq Q'$ is such, and dually in HL. Another important difference is in rule [iter] for Kleene iteration. HL rule $\{\mathsf{iter}\}$ is based on loop invariants, but while these are sound in under-approximation, they are not complete because they don't account for invariants that are reached after a number of iterations [OHe20, §4].

Just like HL, this set of rule is sound and complete for IL:

Theorem 2.7 (IL is sound and complete [OHe20])**.** *An IL triple is provable if and only if it is valid:*

$$\vdash [P] \ \mathsf{r} \ [Q] \iff \vDash [P] \ \mathsf{r} \ [Q]$$

The following example compares HL and IL on a simple program to give an intuition on how they work and differs.

Example 2.8. HL and IL aim at addressing different properties. To show their differences, we use the simple, nondeterministic, terminating program $\mathsf{r}42$:

```
x := nondet();
if (even(x) && odd(y)) {
    z := 42;
}
// assert(z != 42)
```

where we assume that $Q_{42} \triangleq (z = 42)$ denotes the set of erroneous states, i.e., the incorrectness specification. The valid HL triple $\{\text{odd}(y)\} \ \mathsf{r} \ \{Q_{42}\}$ identifies input states that will surely end up in an error state, while the triple $\{\neg Q_{42} \wedge \neg\text{odd}(y)\} \ \mathsf{r} \ \{\neg Q_{42}\}$ characterize input states that will not produce any error.

On the other hand, the valid IL triple $[z = 11] \ \mathsf{r} \ [Q_{42} \wedge \text{odd}(y) \wedge \text{even}(x)]$ expresses the fact that error states in Q_{42} are reachable by safe initial state. Similarly, also the IL triple $[\mathbf{true}] \ \mathsf{r} \ [\neg Q_{42} \wedge \neg(\text{even}(x) \wedge \text{odd}(y))]$ is valid since the postcondition $\neg Q_{42}$ can be reached only when the path conditions to reach the assignment are not satisfied. ■

Remark 2.9 (ok/er flags). One distinguishing feature of proof systems for incorrectness (e.g., [OHe20; Raa+20; RBDO22; RVBO23]) is to tag postconditions, but not preconditions, with either the flag *ok* or *er* to separate successful computations from those leading to errors. An immediate consequence is that the number of proof rules is increased by the necessity to deal with such tags and that the definition of the semantics becomes longer and more complex. Striving for simplicity, in this thesis we mostly follow the alternative from [BGGR23], where the whole concrete domain is extended with such flags, e.g., we use $C = \mathcal{P}(\{ok, er\} \times \Sigma)$ instead of $\mathcal{P}(\Sigma)$, and it is also tacitly assumed that error states are preserved by all transfer functions, i.e., that $[\![\mathsf{r}]\!](er : \sigma) = er : \sigma$, for any $\mathsf{r} \in \mathsf{Reg}$ and $\sigma \in \Sigma$. This way, we accommodate for both pre and post that are tagged, but the treatment of tags is transparent to the rules of the logic. Therefore, in this extended setting, when we write P and Q we leave implicit that they may contain both kinds of tagged states. The distinction between successful and erroneous outputs will be explicitated in the analysis presented in Section 5.4.6.

2.6 Necessary Conditions

The notion of Necessary Conditions (NC) was introduced in [CCL11; CCFL13] for contract inference. The goal was to relax the burden on programmers: while sufficient conditions require the caller of a function to supply parameters that will never cause an error, NC only prevents the invocation of the function with arguments that will inevitably cause an error. Intuitively, given a correctness specification Q, the NC triple

$$(P)\ \mathsf{r}\ (Q)$$

means that any state σ that admits at least one non-erroneous execution of the program r is in P.

Following the original formulation [CCFL13], we can partition the traces of a nondeterministic execution starting from a memory σ in three different sets: $\mathcal{T}(\sigma)$, those without errors, $\mathcal{E}(\sigma)$, those with an error, and $\mathcal{I}(\sigma)$, those which do not terminate. A sufficient precondition $\overline{P}$ is such that $(\sigma \in \overline{P}) \implies (\mathcal{E}(\sigma) = \emptyset)$, that is, $\overline{P}$ excludes all error traces. Instead, a necessary precondition $\underline{P}$ is a formula such that $(\mathcal{T}(\sigma) \neq \emptyset \vee \mathcal{I}(\sigma) \neq \emptyset) \implies (\sigma \in \underline{P})$, which is equivalent to

$$(\sigma \notin \underline{P}) \implies (\mathcal{T}(\sigma) = \mathcal{I}(\sigma) = \emptyset).$$

In other words, a necessary precondition *rules out no good run*: when it is violated by the input state, the program has only erroneous executions. Note that we consider infinite traces as good traces. We do this by analogy with sufficient preconditions, where bad traces are only those which end in a bad state.

Example 2.10. Consider a variation of the program in Example 2.8 that introduces nondeterminism:

```
x := nondet();
if (x is even) {
    if (y is odd) {
        z := 42;
    }
}
// assert(z != 42)
```

As in the previous example, error states are those satisfying $(z = 42)$. Therefore, we consider the *correctness* specification $(z \neq 42)$. Then

	$z = 42$	$z \neq 42$	$z \neq 42 \wedge \text{even}(y)$
$\mathcal{T}(\sigma)$	$\emptyset$	$\neq \emptyset$	$\neq \emptyset$
$\mathcal{E}(\sigma)$	$\neq \emptyset$	$\neq \emptyset$	$\emptyset$

The weakest sufficient precondition for this program is $\overline{P} = (z \neq 42 \wedge \mathrm{even}(y))$ because no input state σ that violates $\overline{P}$ is such that $\mathcal{E}(\sigma) = \emptyset$. On the contrary, we have, e.g., that $\underline{P} = (z \neq 42)$ is a necessary precondition, while $(z > 42)$ is not, because it excludes some good runs. ■

2.7 Separation logic

Separation Logic (SL) [Rey02; ORY01] is an extension of HL introduced to handle pointers and dynamic memory management. Its ingenuity lies in the assertion language used to specify heaps in pre and postconditions: the key insight is that logical conjunction is not apt to describe pointers because of aliasing, therefore the need for a new kind of conjunction. Consider for instance the simple program

$$\mathtt{r} \triangleq \mathtt{*x\ :=\ 1;\ *y\ :=\ 2;\ *z\ :=\ 3}$$

where `x`, `y` and `z` are pointers. The HL triple $\{\mathbf{true}\}\ \mathtt{r}\ \{x \mapsto 1 \wedge y \mapsto 2 \wedge z \mapsto 3\}$ is not valid because the pointer may be aliased at the entry point of the program. To avoid aliasing, the precondition should explicitly state that the addresses are different adding $x \neq y \wedge x \neq z \wedge y \neq z$, which becomes unfeasible as the program grows. This aliasing problem makes also hard to reuse specifications, since variables modified internally by some function calls may be implicitly exposed to the outside through aliasing. To address this problem, SL introduces the separating conjunction $*$ (read "and separately"). To understand how it works, consider the simple formula $x \mapsto 1 * y \mapsto 2$. A heap satisfies this formula if it can be split in two disjoin sub-heaps such that one satisfies $x \mapsto 1$ and the other $y \mapsto 2$. Therefore, this implicitly means that x and y can't be aliased: if they were, the unique memory location couldn't be put at the same time in both disjoint sub-heaps for the two assertions. Intuitively, whenever there is a separating conjunction, every heap location is "owned" by exactly one of the two subformulae involved. Note that SL does not always prevent aliasing: it is always possible to write $x \mapsto 1 * x = y$. The key difference is that here the aliasing is *explicit* in the formula instead of being implicit in the model.

Formally, SL is a triple-based program logic analogous to HL, which uses as assertion language not first-order logic or sets of states but rather formulae built with the following grammar

$$p, q ::= \mathbf{true} \mid p \wedge q \mid \neg p \mid \exists x.p \mid \mathtt{a} \asymp \mathtt{a} \mid \mathbf{emp} \mid \mathtt{a} \mapsto \mathtt{a} \mid x \not\mapsto$$

The first construct describe standard first-order logic. The last three describes heap: **emp** is satisfied only by the empty heap, $\mathtt{a}_1 \mapsto \mathtt{a}_2$ is satisfied by heaps with a single location, corresponding to the evaluation of $\mathtt{a}_1$ and containing the value $\mathtt{a}_2$, and $*$ is the separating conjunction described above.

t do I want to
y about SL?

2.8 Kleene algebras

In this Section, we present Kleene algebras as an algebraic formulation that is able to describe programs.

Definition 2.11 (Idempotent semiring)**.** An idempotent semiring is an algebraic structure $(A, +, \cdot, 0, 1)$ satisfying

- $(A, +, 0)$ is a commutative and idempotent monoid

- $(A, \cdot, 1)$ is a monoid
- multiplication distributes over sum
- 0 is an annihilator for sum, that is $a \cdot 0 = 0 \cdot a = 0$

Intuitively, elements of the idempotent semiring are programs, and the operations are way to compose them: $+$ is nondeterministic choice ($\oplus$ in the syntax of regular commands), $\cdot$ is sequential composition (;), 1 is a no-op (`skip`) and 0 is divergence (the test **false**?).

In any idempotent semiring we define the natural partial order $a \leq b$ iff $a + b = b$. With this definition, $+$ defines the lub of two elements: $a \vee b = a + b$. Moreover, sum and product are monotone in both arguments, and 0 is the bottom element.

To get a Kleene algebra, we need to add the Kleene star operator to model iteration:

Definition 2.12 (Kleene algebra)**.** A Kleene algebra is an idempotent semiring with an additional operator $\star$ satisfying the following axiom:

$$\begin{array}{lll} 1 + a \cdot a^\star = a^\star & 1 + a^\star \cdot a = a^\star & (\textit{Star unfold}) \\ b + a \cdot c \leq c \implies a^\star \cdot b \leq c & b + c \cdot a \leq c \implies b \cdot a^\star \leq c & (\textit{Star induction}) \end{array}$$

(Star unfold) axioms describes that $\star$ behaves as nondeterministic iteration, given it satisfies the same recursive equation. *(Star induction)* axioms are the induction principle for Kleene star. Consider b as the base case, a as the 'increment" operation, c as the inductive thesis and $a \cdot c$ as the inductive step (since it is the "increment" applied to the inductive hypothesis). The axiom says that if we prove that both the base case c and the inductive step $a \cdot b$ are below (ie. satisfy) the inductive thesis b (since we prove their lub, that is their sum, is below b), we proved it for any number of iterations of the increment a starting from the base case b.

Kleene algebras are a versatile formalism that has been applied as is. However, the main algebraic structure we consider is that of Kleene algebra with tests (KAT for short), since this addition allows to encode programs [Koz97].

Definition 2.13 (Test)**.** A test p in an idempotent semiring is an element with a complement $\neg p$ satisfying $p + \neg p = 1$ and $p \cdot \neg p = \neg p \cdot p = 0$. We denote by $\text{test}(A)$ the set of tests of an idempotent semiring A.

With this definition, $(\text{test}(A), +, \cdot, \neg, 0, 1)$ is a boolean algebra contained within A. This means that $+$ represent logical disjunction, $\cdot$ conjunction, 0 false and 1 true.

An interesting example of KAT are so-called "relational KAT".

Example 2.14 (Relational KAT). A relational KAT is a KAT with carrier $\mathcal{P}(C \times C)$, the binary relations on a given set C. $+$ is defined as union and $\cdot$ as sequential composition of relations:

$$a \cdot b = \{(x, z) \mid \exists y \,.\, (x, y) \in a, (y, z) \in b\}$$

0 is the empty relation, 1 is the identity relation, $\star$ is reflexive and transitive closure. A test in this KAT is a subset of 1, that is for any subset $P \subseteq C$ we have the test $\{(x, x) \mid x \in P\}$. Intuitively, this test represent the property P. Nonetheless, when interpreted as an element of the KAT (ie. a command) it behaves as an "assume(P)" statement, or P?, which "filters" states satisfying P and diverges on all other. ■

If C is the set of program states, the corresponding relational KAT can encode programs just like regular commands. Basic expressions are elements of (a subset of) $\mathcal{P}(C \times C)$. Encoding of imperative constructs such as `if` and `while` are just as for regular commands.

$$\frac{}{\{p\}\ 0\ \{1\}}\ \{\mathsf{zero}\} \qquad \frac{}{\{p\}\ 1\ \{p\}}\ \{\mathsf{one}\}$$

$$\frac{p \cdot a = p \cdot a \cdot q}{\{p\}\ a\ \{q\}}\ \{\mathsf{atom}\} \qquad \frac{p \le p' \quad \{p'\}\ a\ \{q'\} \quad q' \le q}{\{p\}\ a\ \{q\}}\ \{\mathsf{cons}\}$$

$$\frac{\{p\}\ a\ \{r\} \quad \{r\}\ b\ \{q\}}{\{p\}\ a \cdot b\ \{q\}}\ \{\mathsf{seq}\} \qquad \frac{\{p\}\ a_1\ \{q\} \quad \{p\}\ a_2\ \{q\}}{\{p\}\ a_1 + a_2\ \{q\}}\ \{\mathsf{choice}\}$$

$$\frac{\{p\}\ a\ \{p\}}{\{p\}\ a^\star\ \{p\}}\ \{\mathsf{iter}\}$$

Figure 2.4: KAT encoding of Hoare logic

Since we can describe programs in KATs, it is natural to ask ourselves whether we can talk about program properties, too. It turns out this is the case, as we can formulate HL in a KAT [Koz00]. Given that we are able to encode program properties (as tests) and programs (as general elements of the algebra), the remaining question is how do we encode the validity of an HL triple. In relational KATs, where we can talk about strongest postcondition, $\mathbf{sp}(a, p) \le q$ is equivalent to the equation $p \cdot a \cdot (\neg q) = 0$. Intuitively, this says that if we start from p, apply a and test for the negation of q, we don't get anything, meeting the intuition that $\mathbf{sp}(a, p)$ is contained in q. A provably equivalent (in any KAT) formulation is the equation $p \cdot a = p \cdot a \cdot q$, that intuitively means that testing for q after applying a on p is redundant. Given the equivalence in relational KATs, it seems reasonable to take this equation as validity for an HL triple in general KATs.

The HL proof system embedded in KATs is shown in Figure 2.4. It handles triples of shape $\{p\}\ a\ \{q\}$, with $p, q \in \text{test}(A)$ and $a \in A$. It is very similar to Figure 2.2 thanks to regular commands and KATs being syntactically very close. The first two rules are subsumed by {atom}, but we prefer to show them explicitly since 0 and 1 are part of the syntax of KATs. Rule {atom} is analogous to the homonymous HL rule. It requires explicitly to check the validity condition since there is no equivalent of the semantics in a KAT. All the remaining rules are the same as HL, just using the syntax of KAT instead of regular commands.

2.9 Abstract interpretation

2.9.1 Abstract domains

Abstract interpretation [CC77; CC79] is a general framework to define static analyses sound by construction, with the main idea of approximating the program semantics on some abstract domain A instead of working on the concrete domain C. The main tool used to study abstract interpretation are Galois connections.

Definition 2.15 (Galois connection)**.** Given two posets C and A, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ define a Galois connection when

$$\forall c \in C, a \in A. \quad \alpha(c) \le_A a \iff c \le_C \gamma(a)$$

that we write $\langle C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \rangle$.

We call C and A the concrete and the abstract domain respectively, α the abstraction function and γ the concretization function. α and γ are also called adjoints. We recall some properties of Galois connections:

Proposition 2.16. *Let $\langle C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A\rangle$ be a Galois connection. Then*

1. $id_C \leq \gamma\alpha$
2. $\alpha\gamma \leq id_A$
3. *α is additive and γ is co-additive*

A concrete value $c \in C$ is called *expressible* in A if $\gamma\alpha(c) = c$. We mostly consider Galois connections in which $\alpha\gamma = \mathrm{id}_A$, called Galois insertions. In a Galois insertion α is onto and γ is injective. A Galois insertion is said to be trivial if A is isomorphic to the concrete domain or if it is the singleton $\{\top_A\}$.

To give an intuition of the role of Galois connections in program analysis, we present the following example.

Example 2.17 (Intervals). Consider as C the set of possible values of a variable, for instance `i`. Since this is an integer value, elements of C are subsets of $\mathbb{Z}$, so $C = \mathcal{P}(\mathbb{Z})$, with the ordering given by set inclusion. A is the set of abstract properties we track in our analysis, and in this example we consider the set of intervals to which `i` may belong. This means

$$A = \mathrm{Int} = \{[n, m] \,|\, n \in \mathbb{Z} \cup \{-\infty\}, m \in \mathbb{Z} \cup \{+\infty\}, n \leq m\} \cup \{[+\infty, -\infty]\}$$

This defines the well known abstract domain of intervals [CC77]. α is the function that abstracts a set S of possible values of `i` to the best (ie. most precise) abstract property:

$$\alpha(S) = [\min(S); \max(S)]$$

with the usual conventions for min and max of empty/unbound set. Since no smaller interval can describe the set S, and this is a superset of S, $\alpha(S)$ is exactly the best abstraction of S.

The concretization γ is the function that does the inverse operation: given an interval $[n, m]$, thought as formal writing or machine representation, gives back its “meaning”, that is the largest subset of $\mathbb{Z}$ that matches that property:

$$\gamma([n, m]) = \{x \in \mathbb{Z} \,|\, n \leq x \leq m\}$$

that is exactly what is commonly represented with $[n; m]$: γ is simply translating the formal writing (or, in our context, an abstract property) to a semantic set of values. We omit for simplicity the cases for infinite ends, as they are as expected.

With these definition, it is straightforward to check that these two functions define a Galois connection (actually, a Galois insertion). ■

We overload the symbol A to denote also the function $\gamma\alpha : C \to C$: this is always an upper closure operator, that is a monotone, increasing (i.e. $c \leq A(c)$ for all c) and idempotent function. In the following, we use closure operators as much as possible to simplify the notation. Particularly, they are useful to denote domain refinements, as exemplified in the next paragraph. Note that they are still very expressive for a Galois insertion (where γ is injective): for instance $A(c) = A(c')$ if and only if $\alpha(c) = \alpha(c')$. Nonetheless, the use of closure operators is only a matter of notation and it is always possible to rewrite them using the adjoints, as the two formulation are equivalent [CC79].

The image of an upper closure operator is a Moore family, that is a subset of C containing the top element $\top_C$ and that is closed under meet. Given a subset $S \subseteq C$, we define its Moore-closure (or meet-closure) as

$$\mathcal{M}(S) = \{\bigwedge X \mid X \subseteq S\}$$

It is well known that any Moore family M is the image of a suitable upper closure operator ρ_M, defined as

$$\rho_M(x) = \bigwedge\{y \in M \mid x \leq y\}$$

and that this defines a Galois connection $\langle C \underset{\rho_M}{\overset{\mathrm{id}_M}{\leftrightarrows}} M\rangle$.

We use $\mathrm{Abs}(C)$ to denote the set of abstract domains over C, and we write $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ when we need to make the two maps α and γ explicit (we omit them when not needed). Given two abstract domains $A_{\alpha,\gamma}, A'_{\alpha',\gamma'} \in \mathrm{Abs}(C)$ over C, we say A' is a *refinement* of A, written $A' \preceq A$, when $\gamma(A) \subseteq \gamma'(A')$. When this happens, the abstract domain A' is more expressive than A, and in particular for all concrete elements $c \in C$ the inequality $A'(c) \leq_C A(c)$ holds. This define a partial order on $\mathrm{Abs}(C)$, and when C is a complete lattice the resulting structure is known to be a complete lattice too [CC79].

A particular kind of domain refinements are pointed refinement [BGGR22]. They are defined adding a single point to the abstract domain, and then performing Moore closure to recover an abstract domain.

Definition 2.18 (Pointed refinement [BGGR22])**.** Let $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ be an abstract domain, and let $z \in C$ be a concrete point. Then the pointed refinement A_z is defined as

$$A_z = \mathcal{M}(\gamma(A) \cup z)$$

We remark that, for any z, $A_z \preceq A$ and, in particular, the two are related by

$$A_z(c) = \begin{cases} A(c) \wedge z & \text{if } c \leq z \\ A(c) & \text{otherwise} \end{cases}$$

2.9.2 Abstracting functions

Since the main object of our study are program semantics, viz. functions, it is important to have the notion of abstraction of a function.

Definition 2.19. Given a monotone function $f : C \to C$ and an abstract domain $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$, a function $f^\sharp : A \to A$ is a *sound approximation* (or abstraction) of f if

$$\alpha f \leq f^\sharp \alpha$$

The *best correct approximation* (bca for short) of f is $f^A = \alpha f \gamma$.

The bca of f is the most precise of all the sound approximations of f: a function $f^\sharp$ is a sound approximation of f if and only if $f^A \leq f^\sharp$.

Through abstraction, it may very well happens that we lose precision, as shown by the following example.

Example 2.20. Consider the interval domain Int of Example 2.17 and the function f given by (the lifting of) the absolute value:

$$f(S) = \{|x| \mid x \in S\}$$

$$\begin{aligned}
[\![\mathsf{e}]\!]^\sharp_A a &\triangleq [\![\mathsf{e}]\!]^A a = \alpha(\!|\mathsf{e}|\!)\gamma(a) \\
[\![\mathsf{r}_1; \mathsf{r}_2]\!]^\sharp_A a &\triangleq [\![\mathsf{r}_2]\!]^\sharp_A [\![\mathsf{r}_1]\!]^\sharp_A (a) \\
[\![\mathsf{r}_1 \oplus \mathsf{r}_2]\!]^\sharp_A a &\triangleq [\![\mathsf{r}_1]\!]^\sharp_A a \vee A [\![\mathsf{r}_2]\!]^\sharp_A a \\
[\![\mathsf{r}^\star]\!]^\sharp_A a &\triangleq \bigvee_{n \geq 0} ([\![\mathsf{r}]\!]^\sharp_A)^n a
\end{aligned}$$

Figure 2.5: Abstract semantics of regular commands.

Its bca is $f^A = \alpha f \gamma$. However, even though this is the most precise abstraction of f we can consider in Int, on some concrete points it still loses precision: fixing as input $S = \{-1, 1\}$ we have

$$\begin{aligned}
\alpha(f(\{-1,1\})) &= \alpha(\{1\}) = [1;1] \\
f^A \alpha(\{-1,1\}) &= f^A([-1;1]) = [0;1]
\end{aligned}$$

■

This issue is well known in abstract interpretation, and for this reason the definition of (global) *completeness* was given [CC79; GRS00].

Definition 2.21 (Complete abstraction)**.** A sound abstraction $f^\sharp$ of f is *complete* when

$$\alpha f = f^\sharp \alpha$$

Intuitively, completeness means that the abstract function $f^\sharp$ is as precise as possible in the given abstract domain A, and in program analysis this allows to have greater confidence in the alarms raised. It turns out that there exists a complete abstraction of f if and only the bca f^A is complete, and if this happens we say that A is complete for f and write $\mathbb{C}^A(f)$. Moreover, since A is complete for f if and only if $\alpha f = f^A \alpha = \alpha f \gamma \alpha$, and since γ is injective (we always assume a Galois insertion), this is true if and only if $\gamma \alpha f = \gamma \alpha f \gamma \alpha$. Recalling that $A = \gamma \alpha$ we define the completeness property $\mathbb{C}^A(f)$ by the equation

$$\mathbb{C}^A(f) \iff Af = AfA.$$

2.9.3 Abstract semantics of regular commands

Consider again regular commands introduced in Section 2.3. While any command r has a bca $[\![\mathsf{r}]\!]^A$, in general an analyser doesn't know it. Instead, it works inductively on the syntax of r to define a sound abstraction of the concrete semantics $[\![\mathsf{r}]\!]$. This (compositional) abstract semantics is given in Figure 2.5, and defines the *abstract interpreter* $[\![\cdot]\!]^\sharp_A : \mathsf{Reg} \to A \to A$. This formulation tells a constructive way to compute an abstract semantics of any regular command r provided that the analyser knows the bca of all expression $\mathsf{e} \in \mathsf{Exp}$ (or at least those appearing in r). This condition could be further relaxed replacing $[\![\mathsf{e}]\!]^A$ with any sound abstraction $[\![\mathsf{e}]\!]^\sharp$ of $[\![\mathsf{e}]\!]$, but for the sake of simplicity we assume the analyser is able to compute all bcas of basic expressions. All other rules are completely analogous to those of the concrete semantics but for the use of the abstract interpreter of syntactic components instead of the concrete semantics. A straightforward proof by structural induction shows that this semantics is monotone and sound:

Proposition 2.22. *The abstract interpreter of Figure 2.5 is monotone and sound w.r.t. the concrete semantics of Figure 2.1; namely for all* $\mathsf{r} \in \mathsf{Reg}$

$$\alpha[\![\mathsf{r}]\!] \leq [\![\mathsf{r}]\!]^{\sharp}_{A}\alpha$$

Even though the abstract interpreter is sound, in general $[\![\mathsf{r}]\!]^{\sharp}_{A} \neq [\![\mathsf{r}]\!]^{A}$, that is the compositional abstraction is less precise than the bca. This is a well-known issue in abstract interpretation, and its main cause is sequential composition. In fact, while in the concrete $[\![\mathsf{r}_1; \mathsf{r}_2]\!] = [\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!]$, in the abstract

$$[\![\mathsf{r}_1; \mathsf{r}_2]\!]^{A} = \alpha[\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!]\gamma \leq \alpha[\![\mathsf{r}_2]\!]\gamma\alpha[\![\mathsf{r}_1]\!]\gamma = [\![\mathsf{r}_2]\!]^{A}[\![\mathsf{r}_1]\!]^{A}$$

that is caused by the fact the abstract interpreter operates on abstract properties only (hence the need for the abstraction between r_1 and r_2).

2.9.4 Under-approximation abstract domains

The definition of Galois connection is not symmetric, in the sense that it puts γ above and α below. This favours over-approximation. A way to see this is the fact that A is an *upper* closure operator, that is $c \leq_C A(c)$. For this reason, to talk about under-approximation domains, we consider "reversed" Galois connections:

Definition 2.23 (Under-approximation Galois connection)**.** Given two posets C and A, a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ define an under-approximation Galois connection when

$$\forall c \in C, a \in A. \quad a \leq_A \alpha(c) \iff \gamma(a) \leq_C c$$

It is important to remark that an under-approximation Galois connection is not a really new notion. In fact, it is just a standard Galois connection between the opposite posets C^{op} and A^{op}. Alternatively, it is a Galois connection between A and C in the reversed order: for this reason, we denote it with $\langle A \overset{\gamma}{\underset{\alpha}{\leftrightarrows}} C\rangle$, where we write A on the left and C on the right. The first point of view allows us to reuse all the machinery for Galois connection, just dualizing results: $A = \gamma\alpha : C \to C$ is a *lower* closure operator (that is monotone, idempotent and reductive, ie. $A(c) \leq_C c$), α is co-additive and γ is additive, the image of A is a dual Moore family (that is a family of sets containing $\bot_C$ and closed under join). Again, we only consider under-approximation Galois insertions (UGI), that are Galois connections in which $\alpha\gamma = \mathrm{id}_A$. We write $A_{\alpha,\gamma} \in \mathrm{Abs}^{\mathrm{op}}(C)$ to say that A is an under-approximation abstract domain on C with adjoints α and γ, possibly omitting these if superfluous.

As an example of under-approximation abstract domain, we propose the following variation of intervals:

Example 2.24. Consider $C = \mathcal{P}(\mathbb{Z})$, while the abstract domain is the set of all intervals (Example 2.17) containing 0, plus the empty interval:

$$\mathrm{Int}_0 = \{I \in \mathrm{Int} \mid 0 \in I\} \cup \{\bot\}$$

where we used $\bot$ to represent the empty interval $[+\infty, -\infty]$. γ is the identity since we want an (under-approximation) Galois insertion, and $\alpha(S)$ is the greatest interval fully contained in S that includes 0. Formally,

$$\alpha(S) = \bigcup\{I \in \mathrm{Int}_0 \mid I \subseteq S\}$$

The result is in Int_0: if it isn't empty, it does indeed contain 0, since is the union of intervals in Int_0 that contains 0 themselves. Moreover it is an interval because union of overlapping intervals is an interval too, and all those intervals intersect at 0. ■

Chapter 3

Related works

This chapter surveys related work that combines over and under-approximations. Section 3.1 analyses KATs as a possible formalism to describe together over and under-approximation, represented respectively by HL and IL. Section 3.2 present LCL_A, a logic that exploits under-approximations to ensure precision of an (over-approximating) abstraction and is able to prove both correctness and incorrectness at once. Section 3.3 discusses `ic3`, an algorithm that combines over and under-approximations in a non-trivial way, and its generalizations. Section 3.4 summarizes Outcome Logic, a triple based program logic that can express both over and under-approximation properties.

3.1 Unifying formalism

Hoare and Incorrectness Logic operate on dual principles. Because of this, some properties are shared among the two, others are dualized and others still are just different. In order to highlight these similarities and differences, it is interesting to embed HL and IL in a common formalism. As recalled in Section 2.8, KATs are able to encode regular commands [Koz97] as well as HL [Koz00] making it a purely equational theory. It would be interesting to do the same for IL, and in principle it seems reasonable to do so given the duality with HL. However, the symmetry is at the level of images of functions (precisely, strongest postconditions), something that is not explicit in the syntax of KAT. Therefore, it is interesting to study which additional algebraic properties are needed to encode IL in KATs.

In [MOH21], the authors propose modal KATs as the diamond modality can be used to represent strongest postcondition [DMS06]. This way, they are able to encode all IL rules but the infinitary [iter], that is needed for completeness of the proof system[1] but requires the existence of countable joins of tests. The authors add this requirement explicitly to the algebraic structure, defining countably-test complete (CTC for short) modal KATs.

Definition 3.1 (Backward diamond)**.** A backward diamond modality on a KAT A is an operator $\langle\cdot| : A \to \text{test}(A) \to \text{test}(A)$ satisfying, for all $a, b \in A$, $p, q \in \text{test}(A)$

$$\langle a|p \leq q \iff p \cdot a \leq a \cdot q$$

$$\langle a \cdot b|p = \langle b|(\langle a|p)$$

[1] Kleene star can be handled using finite unrolling, as we will show for SIL and IL in Figure 5.5. While this approach isn't complete, it is sufficient to prove many triples. We refer the reader to [MOH21, Figure 2] for finitary rules dealing with Kleene star in KATs.

$$\frac{}{[p]\ 0\ [0]}\ \textsf{[divergence]} \qquad \frac{}{[p]\ 1\ [p]}\ \textsf{[skip]}$$

$$\frac{}{[p]\ a\ [\langle a|p]}\ \textsf{[atom]} \qquad \frac{p \geq p' \quad [p']\ a\ [q'] \quad q' \geq q}{[p]\ a\ [q]}\ \textsf{[cons]}$$

$$\frac{[p]\ a\ [r] \quad [r]\ b\ [q]}{[p]\ a \cdot b\ [q]}\ \textsf{[seq]} \qquad \frac{\exists i \in \{1,2\} \quad [p]\ a_i\ [q]}{[p]\ a_1 + a_2\ [q]}\ \textsf{[choice]}$$

$$\frac{[p_1]\ a\ [q_1] \quad [p_2]\ a\ [q_2]}{[p_1 \vee p_2]\ a\ [q_1 \vee q_2]}\ \textsf{[disj]} \qquad \frac{\forall n \geq 0\,.\, [p_n]\ a\ [p_{n+1}]}{[p_0]\ a^\star\ [\bigvee_{n\geq 0} p_n]}\ \textsf{[iter]}$$

Figure 3.1: CTC modal KAT encoding of IL, from [MOH21]

Intuitively, $\langle a|p$ is the strongest postcondition of a on input p. This correspondence is exact in relational KATs, and so it is natural to use it in all modal KATs. Moreover, the standard soundness of HL in KATs is defined by the equation $p \cdot a = p \cdot a \cdot q$, that can be proved equivalent to $\langle a|p \leq q$ in a modal KAT. This further justifies the usage of backward diamond as strongest postcondition, given that $\mathbf{sp}(a, P) \leq Q$ is the classical soundness condition of HL.

Since a CTC modal KAT satisfies all the KAT axioms, it has for free the standard encoding of HL. IL embedding in CTC modal KAT is detailed in Figure 3.1, taken from [MOH21]. The rules are close to standard IL (Figure 2.3), replacing the semantics $\llbracket \mathsf{e} \rrbracket$ with the backward modality $\langle a|$, sequential composition with $\cdot$ and choice with $+$. This is thanks to closeness of regular command to the syntax of KATs. The two additional rules for 0 and 1 are, just as for HL, subsumed by [atom], but we add them explicitly since 0 and 1 are part of the syntax of KATs.

In this algebraic setting, the author derive soundness and completeness theorems for both IL and HL, that we summarise below. We remark that soundness and completeness are obtained with respect to a validity notion based on backward diamond, precisely $\langle a|p \leq q$ for HL and $\langle a|p \geq q$ for IL. What is really interesting about those are the hypotheses required by each:

Theorem 3.2 (cf. [MOH21])**.**

1. *HL is sound and complete in any modal KAT.*
2. *IL is sound in any CTC modal KAT without the (Star induction) axiom.*
3. *IL is complete in any CTC modal KAT.*

This theorem is interesting because it shows the symmetry between over and under-approximation is less sharp than it appears at first glance. On the one hand, over-approximation requires the KAT axioms for both soundness and completeness (modality is used to define the validity of a triple, but we can do without changing the notion of validity). The fact that the latter has the same requirements as the former is essentially because loop invariants are both sound *and* complete for over-approximation. On the other hand, for under-approximation no such tool is known, so completeness has stronger hypotheses than soundness. We remark that soundness of IL does not require *(Star induction)*, the KAT rephrasing of loop invariants, basically because to obtain an under-approximation there is no need for the loop fixpoint but it is enough to consider any (finite) number of iterations. This is no longer the case for completeness, since we do need the loop semantics to prove that it is a sound under-approximation of itself. This motivate the need for all the KAT axioms (including *(Star induction)*) in point (3) above.

In CTC modal KAT the authors are also able to formally prove an intuitive connection between HL and IL.

Theorem 3.3 (cf. [MOH21]). *In any CTC modal KAT*

$$\not\vDash \{p\}\ a\ \{q\} \iff \exists p', q' \,.\, p' \leq p, q' \not\leq q, \vDash [p']\ a\ [q']$$

On the one hand, this theorem means that whenever a specification (ie. an Hoare triple) is not met there is a valid incorrectness triple showing a violation of that specification. In the theorem, q' is the violation since $q' \not\leq q$, and is in the post of p because we can prove $[p']\ a\ [q']$ with $p' \leq p$. On the other hand, when the specification is true, we don't need incorrectness reasoning since we already know all valid triples (hence all those we can prove) starting from $p' \leq p$ are bound to have a postcondition q' satisfying the specification q (ie. $q' \leq q$). This theorem states a connection between over- and under-approximation that is obvious in the concrete interpretation with programs and states, but that it is not in a general algebraic model. Being able to prove it means the encoding meets the desired intuition, rising confidence in its correctness. Moreover, this theorem is a first (trivial) example of how over- and under-approximation can positively interact.

To conclude this section, we mention the concurrent work [ZAG22], that addresses the same problem. In this work, the authors show that KAT alone are not enough to express IL (cf. Theorem 1 of their paper). To enrich the algebraic structure, they follow the idea of expressing (to some extent) the codomain, too, and propose to add a top element $\top$ to the KAT, defining the so-called TopKAT. With this addition, given any element a of the TopKAT we have that $\top a$ represents the "codomain" of a. This is formal in relational TopKATs:

Proposition 3.4 (cf. [ZAG22]). *In any relational TopKAT $\mathcal{R}$, for any two elements $p, q \in \mathcal{R}$ and letting cod be the codomain operator of a relation,*

$$\begin{aligned} \top p = \top q &\iff \mathit{cod}(p) = \mathit{cod}(q) \\ \top p \leq \top q &\iff \mathit{cod}(p) \subseteq \mathit{cod}(q) \end{aligned}$$

The authors then use this definition to embed IL in a generic TopKAT. We remark that in this article the focus is not on completeness of the logic, so there isn't the CTC assumption on the algebraic structure, but only the those join needed to apply [iter] are required to exist. Even with this focus shift, the work already shows that a top element is a viable alternative to modality for encoding IL in Kleene algebras.

3.2 Local completeness

Local Completeness Logic on an abstract domain A (LCL$_A$ for short) [BGGR21] is a first example of non-trivial over and under-approximation interaction, with the former embodied by abstract interpretation and the latter by IL.

As described in Section 2.9, abstract interpretation is always sound, but in general it is not complete: composition of best correct abstractions (bcas) is not the bca of the composition. While in theory completeness ensures no precision is lost, it is a very uncommon situation in general. One of the causes is its requirement to hold *for all inputs.* To weaken this condition, in [BGGR21] the authors propose a notion of *local* completeness, that depends on a specific input.

$$\frac{\mathbb{C}^A_P(\llbracket \mathsf{e} \rrbracket)}{\vdash_A [P]\ \mathsf{e}\ [\llbracket \mathsf{e} \rrbracket P]}\ (\textsf{transfer}) \qquad \frac{P' \leq P \leq A(P') \quad \vdash_A [P']\ \mathsf{r}\ [Q'] \quad Q \leq Q' \leq A(Q)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\textsf{relax})$$

$$\frac{\vdash_A [P]\ \mathsf{r}_1\ [R] \quad \vdash_A [R]\ \mathsf{r}_2\ [Q]}{\vdash_A [P]\ \mathsf{r}_1; \mathsf{r}_2\ [Q]}\ (\textsf{seq}) \qquad \frac{\vdash_A [P]\ \mathsf{r}_1\ [Q_1] \quad \vdash_A [P]\ \mathsf{r}_2\ [Q_2]}{\vdash_A [P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \vee Q_2]}\ (\textsf{join})$$

$$\frac{\vdash_A [P]\ \mathsf{r}\ [R] \quad \vdash_A [P \vee R]\ \mathsf{r}^\star\ [Q]}{\vdash_A [P]\ \mathsf{r}^\star\ [Q]}\ (\textsf{rec}) \qquad \frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad Q \leq A(P)}{\vdash_A [P]\ \mathsf{r}^\star\ [P \vee Q]}\ (\textsf{iterate})$$

Figure 3.2: The proof system LCL_A, from [BGGR21].

Definition 3.5 (Local completeness, cf.[BGGR21])**.** Let $f : C \to C$ be a concrete function, $c \in C$ a concrete point and $A \in \mathrm{Abs}(C)$ and abstract domain for C. Then A is *locally complete* for f on c, written $\mathbb{C}^A_c(f)$ iff

$$Af(c) = AfA(c).$$

A remarkable difference between global and local completeness is that, while the former can be proved compositionally on the command only [GLR15], the latter needs the input to each fragment of the program. Consequently, to carry on a compositional proof of local completeness, information on the input to each subpart of the program is also required, i.e., all traversed states are important. However, local completeness enjoys an “abstract convexity” property: if f is locally complete on a point c, then it is locally complete on any point d between c and $A(c)$:

Lemma 3.6 (Abstract convexity, cf. [BGGR21])**.** *If* $\mathbb{C}^A_c(f)$ *and* $c \leq d \leq A(c)$*, then* $\mathbb{C}^A_d(f)$*.*

This observation has been crucial in the design of the proof system LCL_A. The proof system depends on an abstract domain A, and is able to prove triples $\vdash_A [P]\ \mathsf{r}\ [Q]$ ensuring that:

1. Q is an under-approximation of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$,

2. Q and $\llbracket \mathsf{r} \rrbracket P$ have the same over-approximation in A,

3. A is locally complete for $\llbracket \mathsf{r} \rrbracket$ on input P.

Point (2) means that, given a specification *Spec* expressible in A, any provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ either proves correctness of r with respect to *Spec* or expose an alert in $Q \setminus \mathit{Spec}$. This in turns correspond to a true one because Q is an under-approximation of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$, as pointed out by Corollary 3.8 below.

The proof system is defined in Figure 3.2. It is a logic of under-approximations, much like IL (actually IL is a special case of LCL_A), but with one additional constraint: the under approximation Q must have the same abstraction of the concrete semantics $\llbracket \mathsf{r} \rrbracket P$, as for instance explicitly required in rule (relax). This, by the abstract convexity property mentioned above, means that local completeness of $\llbracket \mathsf{r} \rrbracket$ on the *under-approximation* P of the concrete store is enough to prove local completeness. This way, LCL_A exploits the interaction of over- and under-approximation. The latter is used to ensure the abstraction is locally complete, ie. guarantees precision of the over-approximation. Conversely, the presence of the abstraction in rule (iterate) speeds up the computation as it allows to stop as soon as $A(P)$ is an abstract loop invariant, not having to deal with (possible infinitary) concrete invariants.

Formally, the three key properties (1–3) above are formalized by the following result:

Theorem 3.7 (Soundness, cf.[BGGR21]). *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. *If* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then:*

1. $Q \leq [\![\mathsf{r}]\!]P$,
2. $\alpha([\![\mathsf{r}]\!]P) = \alpha(Q)$,
3. $[\![\mathsf{r}]\!]^\sharp_A \alpha(P) = \alpha(Q)$.

We say that a triple satisfying these three conditions is *valid*, written $\vDash_A [P]\ \mathsf{r}\ [Q]$. As a consequence of this theorem, given a specification expressible in the abstract domain A, a provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ can determine both correctness and incorrectness of the program r:

Corollary 3.8 (Proofs of Verification, cf. [BGGR21]). *Let* $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$ *and* $a \in A$. *If* $\vdash_A [P]\ \mathsf{r}\ [Q]$ *then*

$$[\![\mathsf{r}]\!]P \leq \gamma(a) \iff Q \leq \gamma(a).$$

The corollary is useful in program analysis and verification because, given a specification expressible in A and a provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$, it allows to distinguish two cases.

- If $Q \subseteq \gamma(a)$, then we have also $[\![\mathsf{r}]\!]P \subseteq \gamma(a)$, so that the program is correct with respect to the specification.
- If $Q \nsubseteq \gamma(a)$, then also $[\![\mathsf{r}]\!]P \nsubseteq \gamma(a)$, that means $[\![\mathsf{r}]\!]P \setminus \gamma(a)$ is not empty and thus contains a true alert of the program. Moreover, since $Q \subseteq [\![\mathsf{r}]\!]P$ we have that $Q \setminus \gamma(a) \subseteq [\![\mathsf{r}]\!]P \setminus \gamma(a)$, so that already Q is able to pinpoint some issues.

To better show how this work, we briefly introduce the following example (discussed also in [BGGR21] where it is possible to find all details of the derivation).

Example 3.9. Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$, the abstract domain Int of intervals, the precondition $P = \{1; 999\}$ and the command $\mathsf{r} \triangleq (\mathsf{r}_1 \oplus \mathsf{r}_2)^\star$, where

```
r₁ ≜ (x > 0)?; x := x - 1
r₂ ≜ (x < 1000)?; x := x + 1
```

In LCL_A it is possible to prove the triple $\vdash_{\mathrm{Int}} [P]\ \mathsf{r}\ [Q]$, whose postcondition is $Q = \{0; 2; 1000\}$. Consider the two specification $\mathrm{Spec} = (x \leq 1000)$ and $\mathrm{Spec}' = (x \geq 100)$. The triple is then able to prove correctness of Spec and incorrectness of Spec′. For the former, observe that $Q \subseteq \mathrm{Spec}$. By Corollary 3.8 we then know $[\![\mathsf{r}]\!]P \subseteq \mathrm{Spec}$, that is correctness. For the latter, Q exhibits two witnesses to the violation of Spec′, that are $0, 2 \in Q \setminus \mathrm{Spec}'$. By point (1) of soundness we then know that $0, 2 \in Q \subseteq [\![\mathsf{r}]\!]P$ are true alerts. ■

In LCL_A, being able to prove any triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ allows to show both correctness and incorrectness of r. However, if r is not locally complete on P, or more in general any of the local completeness proof obligations introduced by rule (transfer) (the only axiom of the logic) fails, the proof cannot be completed. To handle this issue, [BGGR22] proposes the idea of changing the abstract domain in which the derivation is performed. Following what had been done for completeness [GRS00], they propose to minimally (in the lattice of abstract interpretations) refine the abstract domain. Unluckily, such a minimal refinement in general does not exists, so that the authors propose a different notion of "best" refinement. They consider pointed refinements, that are defined by the addition of a single point to the abstract domain (followed by a Moore closure operation). Then they compare these pointed refinements not in the lattice of abstract interpretations but by the precision of the additional point. When there exists a most abstract point whose pointed refinement is locally complete, they call this domain *pointed (locally complete) shell*:

Definition 3.10 (Pointed shell, cfr. [BGGR22])**.** Let $f : C \to C$ be a monotone concrete function, $A \in \mathrm{Abs}(C)$ be an abstract domain and $c \in C$ a concrete point. The pointed shell of A on c w.r.t. f exists when the maximum of the set

$$\{x \in C \mid \mathbb{C}_c^{A_x}(f)\}$$

exists and, letting u be such maximum, the pointed shell is $A_u \in \mathrm{Abs}(C)$.

Other than characterizing the existence of pointed shell, they propose two strategies to repair the abstract domain using pointed shells. One of the two, the so-called backward repair, mostly operates on the abstract and so doesn't fit our goal of combining over- and under-approximation. The other, forward repair, instead operates on under-approximation of concrete points. It processes local completeness proof obligations in order, starting from the input and following the control flow. Thanks to abstract convexity of local completeness, this strategy works even on under-approximations of concrete stores, so that it integrates well with LCL_A. The strategy computes local completeness proof obligations in order, either reaching the end of the program (thus completing the analysis) or finding a failed one. In this case, it repairs the abstract domain to the pointed shell (using the under-approximation) and then restart the analysis in the refined domain.

To conclude this section, we point out there exists an algebraic formulation of LCL_A. In [MR22], the authors take inspiration from the works discussed in the previous Section 3.1 and embed LCL_A in (a suitable extension of) KAT. Their first contribution is the definition of an abstract interpretation of KAT, a problem not studied before. Exploiting this, they embed LCL_A in both modal KATs and TopKATs. The technical development of these embeddings is similar to that of [MOH21] and [ZAG22], but it shows that such an embedding is effective as it preserves al property of LCL_A. Lastly, we remark that just as LCL_A generalizes IL, in [MR22] they recover the embedding of IL in modal KATs/TopKATs as a special case of LCL_A's.

3.3 IC3/PDR

`ic3` ("Incremental Construction of Inductive Clauses for Indubitable Correctness"), also called PDR ("Property directed reachability"), was first proposed by Bradley as a model checking algorithm [Bra11]. Given a safety property P and a finite transition system, it either proves the property or outputs a counterexample. In this sense, `ic3` operates both as prover and a bug finder [Bra12]. Its ingenuity consists in using an over-approximation to guide the search for counterexamples, that in turn are used to help refining little by little the over-approximation. This means that the core of `ic3` is a combination of over- and under-approximations. Thanks to this, it quickly became one of the best hardware model checker. Moreover, the algorithm was later applied to other settings, such as probabilistic transition systems, software model checking or generic complete lattices.

In its original formulation, `ic3` operates on a finite transition system. Let $\mathcal{S}$ be the (finite) set of states, $I \subseteq \mathcal{S}$ the set of initial states and $T : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ the transition function: given a set of states S, it returns the sets of states reachable from S in one step. If $\to \subseteq \mathcal{P}(\mathcal{S} \times \mathcal{S})$ is the transition relation, $T(S) \triangleq \{t \mid \exists s \in S \,.\, s \to t\}$. In this case, $\mathrm{lfp}(T \cup I)$

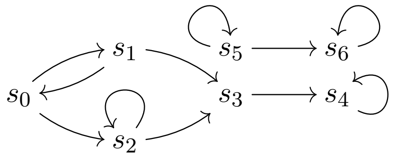


Figure 3.3: The transition system of Example 3.11, with $S = \{s_0, \dots s_6\}$ and $I = \{s_0\}$.

is the set of all states reachable from I. Actually, I and T are represented by propositional formulas and a SAT solver is employed to prove implications and satisfiability queries.

Example 3.11 (Safety problem for transition systems). Consider the transition system in Figure 3.3. Hereafter we write S_j for the set of states $\{s_0, s_1, \dots, s_j\}$ and we fix the set of safe states to be $P = S_5$. We know that the transition system is safe if and only if $\mathrm{lfp}(T \cup I) \subseteq P$, and by Kleene Theorem 2.3 this correspond to the limit of the intial chain

$$\emptyset \subseteq I \subseteq S_2 \subseteq S_3 \subseteq S_4 \subseteq S_4 \subseteq \cdots$$

which stabilizes at S_4 and is therefore below P. Note that the $(j+1)$-th element of the initial chain contains all the states that can be reached from I in at most j transitions.

It is worth to remark that T has a right adjoint $G \colon \mathcal{P}(S) \to \mathcal{P}(S)$ defined for all $X \in \mathcal{P}(S)$ as $G(X) \triangleq \{s \mid \forall t. s \to t \implies t \in X\}$. Thus by (2.1), $\mathrm{lfp}(T \cup I) \subseteq P$ iff $I \subseteq \mathrm{gfp}(G \cap P)$. We can check this by computing the final chain

$$\cdots \subseteq S_4 \subseteq S_4 \subseteq P \subseteq S$$

which again stabilizes at S_4 and is therefore below P. Note that the $(j+1)$-th element of the final chain contains all the states that in at most j transitions reach safe states only. ■

`ic3` fundamental data structures is a sequence $(X_i)_{0 \leq i \leq k+1}$ of over-approximations of states reachable in at most i steps. k is the number of so-called *major iterations* the algorithm has performed. In `ic3`, X_i are logical formulas in CNF. Moreover, the following invariants are kept: (1) $\mathrm{clause}(X_{i+1}) \subseteq \mathrm{clause}(X_i)$, (2) $T(X_i) \Rightarrow X_{i+1}$, (3) $X_k \Rightarrow P$. As a consequence, (1) implies $X_i \Rightarrow X_{i+1}$, and together with (3) this means all elements of the sequence (but possibly X_{k+1}) are strengthening of P. X_k is the "frontier" of the analysis. k is increased by major iterations. Being at major iteration k means the algorithm has proved no violation of P is reachable within k steps, and is working on $k+1$. Once the algorithm can prove $T(X_k) \Rightarrow P$, it refines X_{k+1} by conjoining it with P, increases k and sets $X_{k+2} = \mathbf{true}$ (the empty set of clauses).

At this point, the algorithm also performs what is called clauses propagation. Basically, it considers any clause c in any of the X_i and checks whether $T(X_i) \Rightarrow c$. If this is the case, c is conjoined to X_{i+1}, as this preserves all the invariants. In this sense, the clause c is "propagated" from X_i to X_{i+1}, and the algorithm goes on with other clauses in X_i, then with X_{i+1} and so on. If during this step, at any point $X_i = X_{i+1}$, the algorithm proved the specification P because $T(X_i) \Rightarrow X_{i+1} = X_i$, so X_i is an invariant, and $X_i \Rightarrow P$.

However, in general the implication $T(X_k) \Rightarrow P$ won't be satisfied. In this case, the SAT solver produces a counterexample s, that is a state in X_k such that one of its successors doesn't satisfy P. This means that not only X_k is not an invariant, but that it contains some states which violate (after one step) the specification. The idea of `ic3` is then to see whether this state is introduced by the over-approximation, so that it can be ruled out, or is really reachable, so that a true counterexample is found. So the algorithm looks for a clause c to rule out s from X_k. It takes c whose literals are a subset of those in $\neg s$ (so that $c \Rightarrow \neg s$) and checks whether $I \Rightarrow c$ and $T(X_k \wedge c) \Rightarrow c$. If it can find any such c, it conjoins it to all X_i up to X_{k+1} (it's easy to check this preserves all three invariants). Doing so, the algorithm removes s from X_k, so it tries again the implication $T(X_k) \Rightarrow P$, that will either be satisfied or find a different counterexample. If instead no c satisfies this implication, the algorithm does the same for X_{k-1}, and then for X_{k-2}. Suppose it finds it at this last check (it can't go further because s is not in $T(X_{k-2})$, as it has a successor violating P while $T(X_{k-2}) \Rightarrow X_{k-1}$ and $T(X_{k-1}) \Rightarrow P$, hence $T(X_{k-2}) \Rightarrow \neg s$). So the algorithm found a clause c such that $c \Rightarrow \neg s$, $I \Rightarrow c$ and $T(X_{k-2} \wedge c) \Rightarrow c$: it can conjoin it to all X_i up to X_{k-1}. This rules s out of X_{k-1} but doesn't solve the issue of s being in X_k. However, here's the ingenuity of the algorithm: now the query $T(X_{k-1}) \Rightarrow \neg s$ is either satisfied, so we can conjoin $\neg s$ to X_k, or its failure pinpoints a predecessor t of s in X_{k-1}. That is, it either shows s is a spurious counterexample introduced by the over-approximation of X_k or it traces it back to a possible counterexample in X_{k-1}.

The procedure restarts from t and X_{k-1}. This recursive call will either prove t spurious, so that we can turn back to $T(X_{k-1}) \Rightarrow \neg s$, or find a predecessor u of t in X_{k-2}. This going up and down the sequence of over-approximation, refining it along the way, in the end will either prove $T(X_k) \Rightarrow P$ or find a chain of true counterexample starting from I, disproving P. In this exploration, going up the chain means the over-approximation allows to discard a counterexample, while going down means the counterexample (that is an under-approximation) guides the refinement of the over-approximation.

Our description is high level and leaves out many details. We're not interested in discussing them here, and refer the reader to [Bra11], as we will detail the more general algorithm LT-PDR later in this Section. The one thing we want to point out is that the choice of the clause c used to rule out states violating the specification is minimal, ie. no strict subclause (made of a subset of the literals in c) satisfies the properties required. This in turn means that c includes less states (removing a literal from a disjunction makes it smaller), that is it removes from F_i as much states as possible. This choice is done in order to remove spurious counterexamples as quickly as possible. We will discuss this issue later.

`ic3` was developed for hardware model checking, that means variables are boolean and the transition system is finite. However, its core ideas are deep and not tied to the specific domain, which lead to a host of derived work in other fields. One of the main challenges to generalize it is moving to infinite states space, since termination of the algorithm relies on finiteness of the domain. To cope with this issue, the crucial step is "generalization", that in `ic3` is the choice of $c \Rightarrow \neg s$ to remove s. Taking c as general as possible removes many (unreachable) states at once. Clearly this doesn't impact termination if the state space is finite, but is crucial whenever it is infinite. Hence, while generalization is not needed for soundness, it becomes fundamental for termination of infinite generalizations. This notwithstanding, `ic3` has been successfully generalized. One example is `PrIC3` ("Probabilistic `ic3`") [Bat+20], for model checking Markov decision processes (MDPs). These are basically transition systems where, fixed the action, the state to which the system transitions is not determined but is chosen with a given probability distribution. Without entering in too much details, the key difference is that a system configuration is not a single state but a probability distribution, meaning the state space is infinite. The authors propose a first algorithm which depends on an heuristic for generalization and always terminates, but gives up correctness when the system detects a counterexample: it may very well return a false alarm. Then, to recover correctness, they propose an effective way to find a "good" heuristics, that basically amount to run their algorithm repeatedly, using

```
                         LT-PDR $(F,\alpha)$
<INITIALISATION>
  $(\vec{x}\|\vec{c})_{n,k}$  :=  $(\bot, F\bot\|\varepsilon)_{2,2}$
<ITERATION>                                    % $\vec{X},\vec{C}$ not conclusive
  case :
       $k \geq 2$  :                               %(Induction)
         choose  $j \geq 2$ and $z \in L$ such that $x_j \not\leq z$ and $F(x_{j-1} \wedge z) \leq z$;
         $(\vec{x}\|\vec{c})_{n,k}$  :=  $(\vec{x} \wedge_j z\|\vec{c})_{n,k}$
       $\vec{c} = \varepsilon$ and $x_{n-1} \leq \alpha$  :                 %(Unfold)
         $(\vec{x}\|\vec{c})_{n,k}$  :=  $(\vec{x}, \top\|\varepsilon)_{n+1,n+1}$
       $\vec{c} = \varepsilon$ and $x_{n-1} \not\leq \alpha$  :                 %(Candidate)
         choose  $z \in L$ such that $z \leq x_{n-1}$ and $z \not\leq \alpha$;
         $(\vec{x}\|\vec{c})_{n,k}$  :=  $(\vec{x}\|z)_{n,n-1}$
       $\vec{c} \neq \varepsilon$ and $c_k \leq F(x_{k-1})$  :              %(Decide)
         choose  $z \in L$ such that $z \leq x_{k-1}$ and $c_k \leq F(z)$;
         $(\vec{x}\|\vec{c})_{n,k}$  :=  $(\vec{x}\|z, \vec{c})_{n,k-1}$
       $\vec{c} \neq \varepsilon$ and $c_k \not\leq F(x_{k-1})$  :              %(Conflict)
         choose  $z \in L$ such that $c_k \not\leq z$ and $F(x_{k-1} \wedge z) \leq z$;
         $(\vec{x}\|\vec{c})_{n,k}$  :=  $(\vec{x} \wedge_k z\|\mathsf{tail}(\vec{c}))_{n,k+1}$
  endcase
<TERMINATION>
  if $\exists j \in [0, n-2] \,.\, x_{j+1} \leq x_j$ then return true  % $\vec{X}$ conclusive
  if $k = 1$ then return false                      % $\vec{C}$ conclusive
```

Figure 3.4: LT-PDR, from [Kor+22]

the false counterexample to refine the heuristic every step until either safety is proved or a true counterexample is found. Another generalization is software `ic3`, that uses the same core algorithm to model check software systems [CG12; LNNK20]. Given their state spaces are infinite and variables are not just boolean, software `ic3` relies on SMT instead of SAT solvers. Moreover, they exploit explicitly the control-flow structure of programs, as doing so implicitly has been shown far less effective [CG12]. This makes the algorithms more involved, but shows that `ic3` has indeed the potential to scale to infinite state spaces.

3.3.1 LT-PDR

An interesting point of view is taken by [Kor+22]. In this article, the authors propose a generalization of `ic3` whose only constraint is that the space state is a complete lattice, that they call LT-PDR ("Lattice theoretic property directed reachability"). While LT-PDR is extremely generic and needs heuristics to be instantiated for particular domains, it captures the essence of the algorithm, showing in particular which properties are needed for soundness and termination. In fact, the paper highlights how `ic3` is based on Knaster-Tarski (for proving safety) and Kleene (for counterexample) fixed-point theorems (see Section 2.1). LT-PDR is presented in Figure 3.4. Its notation differs a little from `ic3`. Given a complete lattice L, a monotone function $F : L \to L$ and a property $\alpha \in L$, the goal of the algorithm is to either prove or find a counterexample to $\mathrm{lfp}(F) \leq \alpha$. To encode for instance the original `ic3` problem in this settings, it is sufficient to take $L = \mathcal{P}(\mathcal{S})$, $F(S) = I \cup T(S)$ and $\alpha = P$; it is not hard to verify that this gives exactly the model checking problem. With this in mind, F broadly correspond to the transition relation T; the sequence of over-approximations is $(x_i)_{0 \leq i \leq n}$, while $\vec{c}$ is the so-called *negative* *sequence*, and is more or less the call stack for the recursion on predecessors of error states. Intuitively, the algorithm is building an over-approximating sequence $\vec{x}$ of the fixpoint iterates, hoping to reach a safe abstract fixpoint. However, when the next over-approximating iterate is not safe, it must contain a counterexample, that is then either traced back toward initial states or identified as spurious and hence removed.

The algorithm keeps the following invariants on $\vec{x}$, analogously to `ic3`: (1) $x_i \leq x_{i+1}$, (2) $F(x_i) \leq x_{i+1}$, (3) $x_k \leq \alpha$. **Unfold** increases n, advancing the major iteration. It doesn't perform clause propagation, though: this is the duty of **Induction**. The algorithm is nevertheless sound, as it basically already propagates clauses in **Conflict**. **Candidate** broadly correspond to the case when $T(X_k) \Rightarrow P$ is not satisfied, with a slight difference. Here z is a counterexample to $x_{n-1} \leq \alpha$, not an element of x_{n-2} that is predecessor of a bad state. This difference is insubstantial, though, as the algorithm just recurse on this element going to its predecessors. **Decide** and **Conflict** are, respectively, a new recursive call and the conclusion of a previous one. On the one hand, when c_k, the current predecessor of a bad state, is contained in $F(x_{k-1})$, it means it has a predecessor in the over-approximation. This predecessor is z, and the algorithm goes on recursing on it. On the other hand, if c_k is not in $F(x_{k-1})$ it's a spurious counterexample, introduced by the over-approximation: it is then removed refining x_{k-1} by conjoining with a suitable z that excludes c_k, just like the clause c is conjoined to X_i in `ic3`. The two termination conditions correspond to safety and unsafety respectively: the first one checks whether it reached an invariant (that is $X_{i+1} = X_i$); the second one verifies whether the predecessors reached an initial state (as $F(\bot) = I$ with the given definition of L and F).

The choice of z in **Induction**, **Candidate**, **Decide** and **Conflict** is left unspecified by the algorithm. Save for induction (that is anyway not necessary for neither soundness nor termination), there are canonical choices for z in the rules, but better solutions can be provided by heuristics. This allows LT-PDR to accommodate for other known instances of the algorithm, such as `ic3` and `PrIC3`, just fixing the right lattice and heuristics. While in general LT-PDR doesn't help in this choice, it's able to highlight pros and cons behind them. As briefly remarked above, in general we can look for either a bigger or a smaller z in each of the rules. There are two different kind of choices here: **Induction** and **Conflict**, which pick a z to *refine* the over-approximation, and **Candidate** and **Decide**, whose goal is identify a counterexample. For the latter, a bigger z means that we are possibly examining more counterexamples at once. However, if any of these counterexample is spurious we have to apply **Conflict** to remove that z, and possibly restart with a smaller one containing all the counterexamples we didn't discard with the refinement (that are all the true ones, but possibly also some of the false ones). Moreover, this requires to work on many states (e.g., application of F, that must be exact for the algorithm to work) at once, that may be costly. On the other hand, a smaller z means considering just a few counterexamples. While a lucky choice of such an z may lead very quickly to a counterexample, if the safety property is satisfied this may cause the removal of counterexample one by one, possibly taking a lot of time. We note that `ic3` follows the second path, only examining the single counterexample returned by the SAT solver. Considering instead the z used to refine the over-approximation, there are two conflicting, driving forces guiding the choice. On the one hand, a smaller z removes more counterexamples (this is the path chosen by `ic3` with its generalization to a minimal subclause). On the other hand, a bigger z ensures more abstract over-approximations, yielding less expensive computations and, when there are no counterexamples, a faster fixpoint convergence.

The authors discuss the termination of LT-PDR. As many choices are left unspecified, the best they were able to prove unconditionally is the *existence* of a sequence of choices that make the algorithm ends. On the contrary, to have termination for all possible sequence of choices they need more restrictive conditions:

- the complete lattice L is well-founded
- either $\mathrm{lfp}(F) \not\leq \alpha$ (ie. there is a counterexample) or $\mathrm{lfp}(F) \leq \alpha$ (the property is satisfied) and there are no strictly increasing infinite chains bounded by α

Intuitively, the first condition means that eventually the algorithm proceeds to the next major iteration (i.e., increases n). The second instead limit the number of major iterations to a finite number: either it find a counterexample at some point, or it doesn't but then it can't increase arbitrarily while staying below α.

3.4 Outcome Logic

Outcome Logic (OL) [ZDS23] is a recent triple-based program logic, inspired by IL and its ability to search for true bugs. However, it is based on the key insight that under-approximation (of the program behaviour) and reachability (of true error states) are distinct concepts. To handle them separately, OL starts back from HL and generalizes it to use as assertions not properties of states but properties on an *outcome monoid*, which for instance can be sets of states or probability distributions. This allows OL to unify safety and reachability, as well as over and under-approximation in the same logic.

An OL triple has the familiar shape $\langle P \rangle\ \mathsf{r}\ \langle Q \rangle$. However, P and Q are not assertions on states in Σ. Rather, they are properties over an outcome monoid $M\Sigma$. Particularly, this allows the assertion language to include a new connective $\oplus$, called *outcome conjunction*, that correspond to the monoidal operation on M. To understand the differences, consider the three formulae $x = 0 \wedge y = 1$, $x = 0 \vee y = 1$ and $x = 0 \oplus y = 1$ and the powerset monoid. In this instance, $S \in M\Sigma = \mathcal{P}(\Sigma)$ is a *set of states* rather than just a state $\sigma \in \Sigma$. A set of states S is a model of $x = 0 \wedge y = 1$ if, for all states $\sigma \in S$, $\sigma(x) = 0$ and $\sigma(y) = 1$. This is in line with the intuition we have from HL of this assertion. Things are already a bit different for $x = 0 \vee y = 1$: a set of states S satisfy this formula if either $\sigma(x) = 0$ for all states $\sigma \in S$ or $\sigma(y) = 1$ for all states $\sigma \in S$. Therefore, for instance the set $[x \mapsto 0, y \mapsto 0], [x \mapsto 1, y \mapsto 1]$ does *not* satisfy $x = 0 \vee y = 1$ because it's not the case that neither all states have $x = 0$ nor that all states have $y = 1$. Note that this notion of having all states satisfying either one or the other disjunct has no correspondence in HL because properties are satisfied only by single states, not sets of states. Lastly, the outcome conjunction $x = 0 \oplus y = 1$ is satisfied by a set of states S if it possible to partition $S = S_1 \cup S_2$ in two *non-empty* sets S_1, S_2 such that S_1 satisfies $x = 0$ and S_2 satisfies $y = 1$. For instance, $[x \mapsto 0, y \mapsto 0], [x \mapsto 1, y \mapsto 1]$ does satisfy $x = 0 \oplus y = 1$ because we can split it as $[x \mapsto 0, y \mapsto 0] \uplus [x \mapsto 1, y \mapsto 1]$ where the first set trivially satisfies $x = 0$ and the second $y = 1$. Differently than disjunction though, the set $[x \mapsto 0, y \mapsto 0], [x \mapsto 0, y \mapsto 7], [x \mapsto 0, y \mapsto 42]$ does not satisfy $x = 0 \oplus y = 1$ because you can't split it in two non-empty subsets such that one satisfies $y = 1$: this non-emptiness requirement ensures that all outcomes separated by $\oplus$ are reachable in S.

Given this assertion language, an outcome triple has a validity condition similar to HL. Given a lifting $[\![\mathsf{r}]\!]^\dagger$ of the semantics of r to the monoid $M\Sigma$ (this is, for instance, the Kleisi lifting of $[\![\mathsf{r}]\!]$ when M is a monad) the triple $\langle P \rangle\ \mathsf{r}\ \langle Q \rangle$ is valid if and only, for all elements of the monoid $m \in M\Sigma$ that satisfy the precondition ($m \models P$) then the final outcomes satisfy Q: $[\![\mathsf{r}]\!]^\dagger m \models Q$. For instance, when M is the powerset moand, the lifted semantics $[\![\mathsf{r}]\!]^\dagger$ is simply the collecting semantics from Figure 2.1. This gives a natural way to encode over-approximation (i.e., HL triples), with the addition of the outcome conjunction to prescribe that all outcomes mentioned in the postcondition are truly reachable.

However, the authors want to describe under-approximation as well, that is, be able to specify only a subset of the program behaviours. To achieve this, they use a very specific kind of over-approximation: they introduce a special assertion $\top$ that is satisfied by any element $m \in M\Sigma$. Since all outcomes in the post must be reachable, to specify only some program behaviour instead of all it is enough to over-approximate with $\top$ all the outcomes you don't want to describe. This is a sound over-approximation, but it only specifies that some of the outcomes are truly reachable, thus ignoring some program behaviours and performing under-approximation.

Example 3.12. Consider the motivating example from [ZDS23, § 2.1].

$$\mathsf{r} \triangleq \texttt{x := malloc(); *x := 1}$$

The above program, written in C syntax, allocates a pointer with `malloc` and then tries to dereference it, forgetting that `malloc` can fail and return `null`. Therefore, the HL triple $\{\mathbf{true}\}\ \mathsf{r}\ \{x \mapsto 1\}$ is not valid: if `malloc` fails, the program ends in an error state.[2] The correct HL triple is then $\{\mathbf{true}\}\ \mathsf{r}\ \{(x \mapsto 1) \vee \mathbf{err}\}$, where $\mathbf{err}$ describes that some error occurred. However, this HL triple doesn't tell that both states where $(x \mapsto 1)$ and $\mathbf{err}$ are reachable: they could have been added by the over-approximation.

On the other hand, IL can show reachability of these states: $[\mathbf{true}]\ \mathsf{r}\ [(x \mapsto 1) \vee \mathbf{err}]$ is a valid IL triple. Moreover, we may not be interested in the $x \mapsto 1$ outcomes since we already found an error, and for efficiency reasons a tool may want to drop it. IL accounts for this with its consequence rule, that allows to derive the triple $[\mathbf{true}]\ \mathsf{r}\ [\mathbf{err}]$.

OL is able to do both, via the outcome conjunction $\oplus$ and the trivial outcome $\top$. The OL triple $\langle\mathbf{true}\rangle\ \mathsf{r}\ \langle(x \mapsto 1) \oplus \mathbf{err}\rangle$ is valid, stating the safety property that all reachable outcomes satisfy $(x \mapsto 1) \oplus \mathbf{err}$. However, it also tells a reachability property, namely that both $(x \mapsto 1)$ and $\mathbf{err}$ are reachable outcomes of the program. To account for under-approximation, OL can use its consequence rule to weaken the postcondition according to the implication $(x \mapsto 1) \oplus \mathbf{err} \implies \top \oplus \mathbf{err}$: therefore, the OL triple $\langle\mathbf{true}\rangle\ \mathsf{r}\ \langle\top \oplus \mathbf{err}\rangle$ is valid. Since $\top$ is a trivial assertion, satisfied by any outcome, this triple means that $\mathbf{err}$ is a reachable outcome, and other reachable outcomes (if any) are unconstrained because they only have to satisfy $\top$. ■

In the paper, the authors give a proof system for OL [ZDS23, Figure 4] that is parametric in the chosen outcome monoid. They also show how to enrich this proof system with rules specific for probabilistic programs [ZDS23, Figure 7] and heap-manipulating programs [ZDS23, Figure 6]. This last instance is further explored in subsequent work [ZSS24], which propose Outcome Separation Logic, a logic where the heap manipulation is baked into the logic itself and can be further composed with the outcome monoid to get, for instance, a probabilistic separation OL. This allows to explore a new tri-abduction algorithm and to prove a more general frame rule that is valid for any outcome monoid.[3]

Another result proved for OL is the ability to disprove triples within the logic itself. As show for instance in Theorem 3.3, IL can disprove HL triples, i.e., prove that a given HL triple is not valid. However, neither HL nor IL can disprove triples of the same logic: no (set of) valid HL triple can show that an HL triple is not valid. Instead, OL is able to do just that. If we consider sets of elements of $M\Sigma$ instead of syntactic assertions

[2] This is actually undefined behaviour in C. We won't deal with that here and just assume this actually causes a recognizable error, such as a segmentation fault.

[3] The OL instance already admitted a frame rule but it was limited to deterministic computations.

(see Section 2.4), any OL triple is not valid if and only if some other OL triple is valid. Thus, encoding HL triples as OL ones, it is possible to disprove them within the logic. Moreover, this result is made syntactic (i.e., using formulae in an assertion language) for the nondeterministic and probabilistic instance of OL presented in the paper. Therefore, OL can use under-approximation to disprove over-approximation and vice versa.

We conclude pointing out that [Zil24] propose yet another generalization of OL (using weighted computations) for which they provide a complete proof system, at the cost of using semantics assertions (in that case, arbitrary weighted collections of elements of Σ) instead of syntactic ones.

3.5 Conclusions

In this chapter, we showed some works that exploits both over and under-approximation. They pinpoint symmetries as well as fundamental differences between the two, and combines them so that they help each other, in order to get the best out of both. We discussed an algebraic formulation that incorporate both, then three techniques - namely LCL_A, `ic3`/PDR and OL - which are able to exploit this combination in different and non trivial ways. However, we believe there are other ways to exploit such an interaction, and this is what we are discussing in the next chapters.

Chapter 4

Under-approximation abstract domains

In this chapter, we try to use Abstract Interpretation as a basis for under-approximation analysis. In principle, the over-approximation theory could be dualized in an order-theoretic sense to obtain results for under-approximation. However, in this chapter we show that it's not so simple. Particularly, *basic transfer functions* (the semantics of basic constructs of the language) are not dualized: therefore, the dual of an abstract domain that "behaves well" with respect to basic transfer functions may not enjoy the same property.

We first point out some intuitive reasons that break the symmetry between over and under-approximation. Then, building on these observations, we formally derive some negative results showing that it is not possible to define Galois connection-based under-approximation abstract domains in a large class of instances. More in details, we assume that (i) abstract analyses should return non-trivial results for large classes of programs and (ii) to justify the convenience of the abstract analysis, the abstract domain should be significantly "smaller" than the concrete powerset. Under these assumptions, we prove that there is no under-approximation abstract domain able to analyse programs encoding certain classes of basic transfer functions.

The content of this chapter is based on [ABG22; ABG24].

4.1 Overview

In their first work on Abstract Interpretation [CC77], Patrick and Radhia Cousot introduced the formal theory that could be used to study both over and under-approximations. However, while the former has been extensively studied, there are only sparse studies on under-approximation abstract domain. For instance, Lev-Ami et al. [LSRG07] proposed to use complements of over-approximation domains to infer sufficient preconditions for program correctness. However, such an approach is severely limited in proving incorrectness, as we show in Example 4.1. For the same goal, Miné [Min14] uses directly over-approximation domains, giving up the best abstraction and handling the choice of a maximal one with heuristics. To infer necessary preconditions, Cousot et al. [CCL11; CCFL13] use Abstract Interpretation techniques but on boolean formulas, hence bypassing the issue of defining an under-approximation abstract domain. Schmidt [Sch07] uses higher-order domains, defining abstract states with the meaning "there exists a value satisfying this over-approximation property", hence giving rise to an under-approximation of over-approximations. All the above approaches design under-approximation domains starting from over-approximation ones, and, to the extent of our knowledge, there are no abstract domains thought from the ground up for under-approximation.

We consider the problem of defining meaningful under-approximation abstract domains for program analysis over powerset concrete domains under the hypotheses (i) and (ii) above. From a purely mathematical point of view, this seems a trivial task because the theories of over and under-approximation are dual. For instance, as done by Lev-Ami et al. [LSRG07], we can transform any over-approximation domain into an under-approximation by reversing the order of its elements and complementing their interpretation. We call this construction *complement domain.* As an example, consider the (over-approximation) interval domain, where, e.g., the interval $[-1, 1]$ is a correct abstraction for any subset of $\{-1, 0, 1\}$ and is the best abstraction of $\{-1, 1\}$ and $\{-1, 0, 1\}$. Instead, in the complement domain, the interval $[-1, 1]$ is a correct abstraction of any set containing all values strictly smaller than -1 and all values strictly greater than 1 and is the best abstraction of $\{\dots, -3, -2, 0, 2, 3, \dots\}$ and $\{\dots, -3, -2, 2, 3, \dots\}$. Note that, being an under-approximation, $[-1, 1]$ represents correctly any set *larger* than its concretization $\{\dots, -3, -2, 2, 3, \dots\}$. However, we argue that complement domains are not useful for incorrectness analysis: initializations such as `i := 0` or `i := 1000` are abstracted to the interval $[-\infty, \infty]$, which is the best abstraction of any finite set but loses any information about the initial value of `i`. We give more details on complement domains in Example 4.1.

Another important asymmetry we point out is the handling of divergence. In both over and under-approximation, divergence is represented by the bottom element $\bot$ of the abstract domain. However, $\bot$ as an under-approximation also represents the absence of information; dually, in over-approximation this is described by $\top$. This is a problem since many concrete functions are strict, that is, when applied to a non-terminating expression, they also fail to terminate (they return $\bot$ if one argument is $\bot$), and, to be a correct under-approximation, also the corresponding abstract function needs to be strict:

$$f^\flat(\bot) = f^\flat(\alpha(\emptyset)) \preceq \alpha(f(\emptyset)) = \alpha(\emptyset) = \bot$$

This implies that whenever the analysis cannot determine any meaningful information at some program point, it has to propagate the absence of information along all program paths, at least until a join in the control flow is found. So "recovery" from $\bot$, that is, producing a result different from $\bot$, once we start with it, is very hard in an under-approximation. On the contrary, "recovery" from $\top$ in over-approximation is easier: for example, this can happen whenever the code contains a constant assignment.

A last asymmetry we remark is that over-approximation abstract domains are closed under intersection, while under-approximation abstract domains are closed under union. In the case of assignments this asymmetry has serious consequences. While the result of an assignment can be over-approximated by any larger set of values, with different degrees of precision, the only admissible under-approximations are either the singleton or the empty set. If not enough singletons are represented, then the under-approximation analysis is likely to give a trivial result. Conversely, if too many singletons are represented, closure under union will make the size of the under-approximation abstract domain grow exponentially, violating assumption (ii).

We further strengthen the asymmetry by using the concept of under-approximation Galois insertion (Definition 2.23) to show how straightforward adaptations of some known over-approximation techniques don't work for under-approximation. Then, we establish some negative results. The general theme is to fix some reasonable hypotheses over the common functions encoded by program fragments and then show that any under-approximate abstract domain with size not exponentially larger than the set of values

Result	Concrete domain	Tight hypotheses	Generalizes
Proposition 4.10	$\mathcal{P}(\mathbb{Z})$	–	–
Theorem 4.15	$\mathcal{P}(C)$	High surjectivity	Proposition 4.10
Theorem 4.19	$\mathcal{P}(C)$	High surjectivity	Proposition 4.10

(a) Tabular comparison of our results for infinite domains.

Result	Concrete domain	Tight hypotheses	Generalizes	Inspired by
Proposition 4.13	$\mathcal{P}([-N, N])$	–	–	–
Theorem 4.28	$\mathcal{P}(C)$	None	Proposition 4.13	Theorem 4.19

(b) Tabular comparison of our results for finite domains.

Figure 4.1: Summary of the results in this chapter. The tables acts as a quick reference, showing which concrete domain they are applicable to, which of the hypotheses are (known to be) tight, and the relations between the results.

(i.e., satisfying assumption (i)) will return no useful information for such program fragments. Since the analysis is unable to recover from this lack of information, the result of the analysis of the entire program will be trivial (i.e., violating assumption (ii)). Therefore, while any abstract domain for under-approximate reasoning may be effective for some carefully crafted programs, it will return trivial results on the majority of programs.

Formally, we first introduce the new definition of *non-emptying function* (Definition 4.4), describing functions that do not tamper the analysis. Roughly speaking, a function on the concrete domain is non-emptying if it admits an under-approximation whose consecutive applications would not waste the analysis result by returning $\bot$. Our first result proves that no abstract domain for integers can be constructed that makes all increments non-emptying. In other words, we prove that an analysis based on under-approximation domains would often report trivial information for programs that involve repeated increments. We do so both for the infinite domain $\mathcal{P}(\mathbb{Z})$ and a finite integer domain $\mathcal{P}([-N, N])$ for large N.

We then study how we can generalize these results to different concrete domains and function families. We first focus on the case where the concrete domain is the powerset of an *infinite* set of values, and we prove two results, one local and one global, for infinite concrete domains. To do so, we introduce the notion of *highly surjective function family* (Definition 4.14), of which sums are an instance. The local condition applies to each function in the family, while the global condition is a property of the whole family. Contrary to the definition of non-emptying functions, the notion of highly surjective function family is independent of the abstract domain. Once again the main consequence of our results is that abstract analyses of programs involving the application of functions in the family will often report trivial information. As in the case of increments, highly surjective function families are commonly coded in programs. Finally, we show that the hypothesis of high surjectivity is tight by presenting mathematical constructions of abstract domains making all functions in a family non-emptying. Our results for infinite domains are summarized in Table 4.1a: both results for an arbitrary infinite set C generalize the result for integers. The hypothesis of high surjectivity is tight, but we do not know about all the others.

Lastly, we focus on the powerset of a *finite* set of values. We discuss why a straightforward adaptation of the two results for the infinite case to the finite one was not possible, most notably a difference with the very definition of highly surjective function family. We then propose a general result for this case, inspired by the global condition for the infinite case, but whose details are different. Our results for finite domains are summarized in

Table 4.1b: our single result generalizes again the result for integers, but we were not able to prove any of our hypotheses tight.

4.2 Comparison with over-approximation

We revisit some known over-approximation analyses and techniques, showing how specific characteristics of under-approximation makes this a challenging task.

Complement domain

The first attempt, already briefly discussed in the introduction, is the use of the complement of an over-approximation abstract domain. This is an application of the order theoretic duality, but it turns out the resulting domains are not useful for analysis.

Example 4.1 (Complement domain). Whenever the concrete domain is a powerset $\mathcal{P}(C)$, we can exploit the complement $\neg : \mathcal{P}(C) \to \mathcal{P}(C)$ to define a UGC from any given GC by taking complements of the concretization. Formally, given a GC $\langle \mathcal{P}(C) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A \rangle$, then $\langle \mathcal{P}(C) \underset{\neg\circ\gamma}{\overset{\alpha\circ\neg}{\leftrightarrows}} A^{\mathrm{op}} \rangle$ is a UGC (this stems from $\neg : \mathcal{P}(C) \to \mathcal{P}(C)^{\mathrm{op}}$ being an isomorphism of posets). For instance, given the interval domain, we can define its complement by

$$\gamma_\neg([n, m]) = \neg\gamma([n, m]) = \mathbb{Z} \setminus \{x \in \mathbb{Z} \,|\, n \leq x \leq m\} = \{x \in \mathbb{Z} \,|\, x < n \vee m < x\}$$

The set of abstract elements is the same as Int, but the ordering is reversed, so we call this domain $\mathrm{Int}^{\mathrm{op}}$. The (under-approximation) Galois Connection with $\mathcal{P}(\mathbb{Z})$ is given by $\gamma_\neg = \neg \circ \gamma$ and $\alpha_\neg = \alpha \circ \neg$. Note that, thanks to the ordering in $\mathrm{Int}^{\mathrm{op}}$ being the opposite, both $\alpha_\neg$ and $\gamma_\neg$ are monotone.

While $\mathrm{Int}^{\mathrm{op}}$ is a sound under-approximation abstract domain, we argue that it is not useful for incorrectness analysis. Consider a command as simple as the initialization `i := 0` that may happen at the beginning of a loop. This requires the analysis to abstract the concrete element $\{0\}$, the set of values `i` may assume at the beginning of the loop. According to the above definition, we have

$$\alpha_\neg(\{0\}) = \alpha(\neg\{0\}) = \alpha(\mathbb{Z} \setminus \{0\}) = [-\infty, +\infty]$$

that is the bottom element of $\mathrm{Int}^{\mathrm{op}}$ since the ordering is reversed. We can better understand the reason for getting bottom by recalling that the meaning of an interval in $\mathrm{Int}^{\mathrm{op}}$ is the set of elements that are *not* in the interval:

$$\gamma_\neg([-\infty, +\infty]) = \neg\gamma([-\infty, +\infty]) = \neg\mathbb{Z} = \emptyset$$

Other than the intuition that we lost all the information about the initialization, since $[-\infty, +\infty]$ is $\bot$ the analysis incurs in the issue described in the Introduction about "recover" from $\bot$, effectively making the analysis unable to infer anything.

This line of reasoning can be generalized to any finite concrete set X: $\mathbb{Z} \setminus X$ is not bounded, as it contains all integers greater than $\max(X)$ and smaller than $\min(X)$ (which are both finite), so its abstraction through α is $[-\infty, +\infty]$. ■

Compositionality

An important property of Abstract Interpretation analysis is compositionality. However, citing O'Hearn [OHe20, §8], "for incorrectness reasoning, you must remember information as you go along a path [...]". This means that a compositional under-approximation analysis must be precise enough to have locally all the informations which should be carried over to the next piece of code. Over-approximation instead is way less restricting in this sense, since "for correctness reasoning, you get to forget information as you go along a path" (O'Hearn [OHe20, §8]). As an example, we present dependency analysis, which is compositional for over-approximation but it is not for under-approximation.

Example 4.2 (Dependency analysis). A dependency analysis (e.g., [Ass+17, §6]) aims to compute, for each variable at every program point, the set of values it depends on. They are used for instance to check security properties such as information flow constraints.

The result of the analysis is usually expressed with a collection of atomic dependencies $x \rightsquigarrow y$, meaning that the *current* value of y depends on the *initial* value of x. For instance, consider the code

```
if (x == pwd) { y = y + 100 }; x:= 0
```

The analysis of this fragment returns the dependencies $x \rightsquigarrow y$, $y \rightsquigarrow y$, as the final value of y depends on the initial values of both x and y. Note that the analysis does not compute any dependency with x on the right as the final value of x is always 0, hence it carries no dependency.

Over-approximation dependency analysis heavily exploits *transitivity* (as it enables compositional reasoning): if C_1 exhibit the dependency $x \rightsquigarrow y$ and C_2 exhibits $y \rightsquigarrow w$, then $C_1; C_2$ *may* induce $x \rightsquigarrow w$. As a trivial example, `y := x; w := y` yields the dependency $x \rightsquigarrow w$. However, transitivity is not well-behaved for under-approximation, whose goal is to find true dependencies only. The issue is that dependencies may cancel each other when composed. As a simple example, consider the code

$$C_1 = \texttt{y := x; z := -x}$$
$$C_2 = \texttt{w := y + z}$$

An under-approximation dependency analysis for C_1 may return $x \rightsquigarrow y$ and $x \rightsquigarrow z$ because they are true dependencies. Similarly, for C_2 it could deduce $y \rightsquigarrow w$. However, for the composition $C_1; C_2$ the transitive inference $x \rightsquigarrow y \rightsquigarrow w$ isn't sound because w is always 0, so it doesn't depend on anything. ■

Closure under union

Lastly, we consider non-relational analyses. Intuitively, a non-relational analysis cannot capture relationships between different variables. However, under-approximation cannot forget information along a path, including relations between variables. This means that non-relational domains cannot be used for under-approximation.

Example 4.3 (Non-relational domain). Informally, a non-relational abstract domain is a tuple of elements, one for each variable x, and describes the set of concrete states where each variable belongs to the values in its abstract coordinate. The abstraction is performed on each variable independently, projecting all states in S on that variable and then abstracting the resulting set. The concretization is performed on each variable independently, and then the results are combined in all possible ways to get concrete values.

As an example, consider the product of one interval domains for each variable. For instance, take the code

```
y := 5 - x; z := x + y
```

and assume at the beginning the variable `x` assumes values in the interval $[0, 3]$. An interval analysis on this fragment would find that `y` takes values in the interval $5 - [0, 3] = [2, 5]$, and

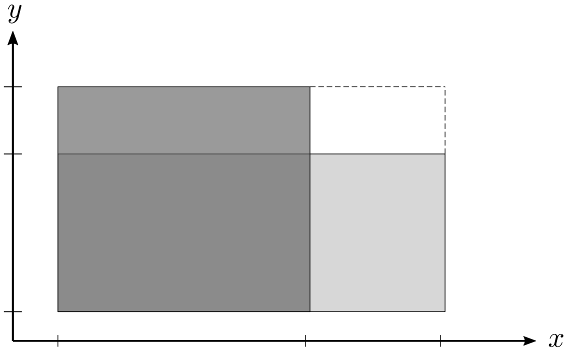


Figure 4.2: Non-relational domains are not closed under union

then `z` is in the result of $[0,3]+[2,5]=[2,8]$. However at the end of the program `z` is always 5, so this is a sound over-approximation but is not as precise as it can be. The issue here is that the values of `x` and `y` are not independent, so an operation between these two variables cannot actually receive all possible inputs with `x` in $[0,3]$ and `y` in $[2,5]$, but just those that also satisfy the *relationship* `y = 5 - x`. However, the interval domain knows nothing about this relationship since it abstracts each variable independently. More precisely, the possible pair of values for x, y are $\{(0,5),(1,4),(2,3),(3,2)\}$, but the abstraction is computed projecting on one variable (for instance x) and then computing the interval over-approximating that set (so that the abstraction for x is $[0,3]$). Then, the concretization contains all the pairs in the product $[0,3]\times[2,5]$, which are much more.

In general, this projection is not sound for under-approximation: the concretization is not able to recover which of the original pairs were in the concrete set and which were not. On a more abstract level, such a domain is not closed under union: elements of the abstract domain are "rectangles" in the Cartesian plane with variables on the axes (since they are concretized to the Cartesian product of the abstractions for each variable) and the union of rectangles is not a rectangle. This is shown in Figure 4.2: the union of the light and dark rectangles is not a rectangle as it misses the top-left "corner". ■

4.3 Non-emptying functions

A central concept in our development is the following definition of *non-emptying function.* To understand the rationale behind it, recall that $\bot$ means no information in the under-approximation setting, while any other abstract value is "something" interesting.

Definition 4.4 (Non-emptying function)**.** Let $\langle C \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} A\rangle$ be a UGC, $f : C \to C$ a monotone function and $f^A = \alpha \circ f \circ \gamma$ its bca. We say that f is *non-emptying* (in A) if, for any concrete value c, $\alpha(c) \neq \bot$ and $\alpha(f(c)) \neq \bot$ imply $f^A(\alpha(c)) \neq \bot$.

The idea behind this definition is that if the analysis starts from something meaningful ($\alpha(c) \neq \bot$) and it can find something significant ($\alpha(f(c)) \neq \bot$) then it will find at least one of the possible results ($f^A(\alpha(c)) \neq \bot$), thus not degrading the result of the analysis to $\bot$ and avoiding the issues discussed above. Intuitively, we want basic transfer functions to be non-emptying so that their analysis doesn't return $\bot$. The following toy example illustrates the meaning of the definition.

Example 4.5. Consider the simple imperative fragment

```
if (x ≠ 0) then { while (x < 10) { y := 7 / x; x := x + 1; } }
```

where a careless programmer used the condition `x` $\neq$ `0` instead of the expected `x > 0`: on any initial state where x is negative the program incurs a division by 0 error.

For the analysis, suppose `x` is an integer value and consider the domain $\mathrm{Int}_{01} = \{I \in \mathrm{Int} \,|\, 0 \in I \vee 1 \in I\} \cup \{\bot\}$, a variation of Int_0 from Example 2.24 such that each interval in Int_{01} must contain at least one of 0 and 1. By an argument similar to that for Int_0 it can be shown that Int_{01} is closed under union (since 0 and 1 are consecutive values in the integer domain), and thus is an under-approximation domain, whose abstraction function maps each set of integers to the largest interval that is included in the set and that contains 0 or 1.

In this domain, the semantics f of the increment $x := x + 1$ is not non-emptying: for instance, on the concrete value $c = \{-1, 1, 2, \ldots, 10\}$ we have $\alpha(c) = [1, 10] \neq \bot$ and $\alpha(f(c)) = \alpha(\{0, 2, 3, \ldots, 11\}) = [0] \neq \bot$ but $f^A(\alpha(c)) = f^A([1, 10]) = \alpha(f(\gamma([1, 10]))) = \alpha(\{2, 3, \ldots, 11\}) = \bot$. We show in the following the effect of this on the analysis.

Assume to start the analysis in this domain with the initial condition $[-1, 10]$ for variable x: remember that this being an under-approximation analysis, the abstract state $[-1, 10]$ means that x may assume all the values in that interval at the beginning of the code fragment. In the concrete execution, the filter $x \neq 0$ then produces the concrete set of values $c = \{-1, 1, 2, \ldots, 10\}$, but the abstract interpreter must abstract this to its largest subset that is an interval containing 0 or 1, that is $[1, 10]$. The abstract analysis of the cycle then proceeds straightforwardly, finding $\bot$ after one iteration of the loop body (since after the increment the set of values for x is $\{2, 3, \ldots, 11\}$ that is abstracted to $\bot$ because it contains neither 0 nor 1) and so the abstract fixpoint of the loop is the interval $[1, 10]$. This yields no error, even though the concrete execution starting at $x = -1$ does indeed fail after one iteration. ■

For the remainder of the paper, we assume to be given a set of *values* C and a UGI $\langle \mathcal{P}(C) \overset{\alpha}{\leftrightarrows} A \rangle$ whose concrete domain is $\mathcal{P}(C)$. In the following, we present the basic technical tools we use to prove our theorems (Propositions 4.10, 4.13 and Theorems 4.15, 4.19, 4.28), which all follows the same pattern. First we make the assumption that the abstract domain is not *too large* to hinder the analysis, but then we show that if all the functions in a given family were non-emptying, the abstract domain would grow too large. Particularly, we start from a representable element (which is assumed to exist by hypothesis), then use Lemma 4.7 to build other representable elements, up until the size of the abstract domain blows up. To get this, we exploit an exponential increase in the size of A caused by "closure under union", that is the fact that if two concrete elements are representable in the abstract domain, their union is representable as well.

Definition 4.6. Let $S \subseteq C$ be a subset of C, and $\langle \mathcal{P}(C) \overset{\alpha}{\leftrightarrows} A \rangle$ is an a UGI. We say that $d \in C$ is *representable with* S if $S \cup \{d\}$ is representable in A. We call $R_A(S)$ the set of elements of C representable with S, i.e.

$$R_A(S) = \{d \in C \,|\, \alpha(\{d\} \cup S) = \{d\} \cup S\}$$

For the sake of brevity, we omit the subscript A whenever it is clear from the context, we write R for $R(\emptyset)$, the set of representable values of C, and use the shorthand $R(c)$ for $R(\{c\})$ when $c \in C$ is a concrete value. The following lemma, valid for non-emptying functions, explains the role played by Definition 4.4 in proving all our negative results. Roughly speaking, it states that for f to be non-emptying, some concrete values must be representable in the abstract domain.

Lemma 4.7. *Let $f : C \to C$ be non-emptying, $c \in R$ and $\tilde{c} \notin R(c)$. If $f(\tilde{c}) \in R$, then $f(c) \in R$.*

While the broad outline of all of our theorems is always the same, they differ in how they solve two key issues: first, it must be possible to apply Lemma 4.7; second, all the new representable elements obtained by applying it must be different from one another. In the following, we present various sets of conditions that can guarantee these two points, hence getting hypotheses for non existence of under-approximation abstract domains.

4.4 Integer domains

In this section, we apply the idea from the previous section on the concrete domain of integers and prove that any under-approximation abstract domain will likely return trivial analyses for programs that include sums inside arithmetic expressions.

4.4.1 Infinite integer domain

As a first example, we consider the infinite domain $\mathcal{P}(\mathbb{Z})$ over all integers.

Assumption 4.8. We assume that an abstract domain A, to be feasible for analyses, must be at most countable.

We make this assumption because we want the analysis to have a complexity comparable to that of a single concrete execution: if the analysis could be as complex as an arbitrary set of concrete executions, we could use those instead of the abstract domain. Therefore, we require the abstract domain to have the same size of the set $\mathbb{Z}$ of values handled by the program, and not the concrete domain $\mathcal{P}(\mathbb{Z})$. Many abstract domains, such as intervals, octagons [Min06] and polyhedrons [CH78] with at most n edges, satisfy it; some, such as general polyhedrons, do not, but indeed they also exhibit a worst-case exponential cost.

Based on Assumption 4.8, we prove a simple cardinality estimate that is exploited to prove that there are few representable elements.

Lemma 4.9. *For any fixed subset $S \subseteq \mathbb{Z}$, $R(S)$ is finite.*

The result for integers now shows that no under-approximation abstract domain makes all sums non-emptying. The idea of the proof is to define an infinite sequence of representable elements, which is in contradiction with the previous lemma that says that $R = R(\emptyset)$ is finite. To define such a sequence, we want to use Lemma 4.7: we start from an initial representable n_0 and from a value $\bar{n}$ not representable with it, then find a non-emptying f that maps $\bar{n}$ into n_0, so that $f(\bar{n})$ is representable and we can then apply the lemma to get the new representable element $f(n_0)$. We then iterate this procedure, changing f, to build the infinite sequence. We believe the hypothesis that there exists an initial representable value is not very restrictive since initializations like `x = 0` must be abstracted to $\bot$ if 0 is not representable.

Proposition 4.10. *Let $\langle \mathcal{P}(\mathbb{Z}) \overset{\alpha}{\leftrightarrows} A \rangle$ be a UGI, and assume that there is an integer n_0 that is representable. Then it cannot be the case that all the functions of the form $f_n(x) = x + n$ are non-emptying in A.*

Proof. Towards a contradiction, let us assume that all f_n are non-emptying in A. By hypothesis, $n_0 \in R$ and $R(n_0)$ is at most finite by Lemma 4.9. Since $\mathbb{Z}$ is infinite, there exists an $\bar{n} \in \mathbb{Z} \setminus (R(n_0) \cup \{n_0\})$, that is $\bar{n}$ is an element such that the pair $\{n_0, \bar{n}\}$ is not representable. Let $d = n_0 - \bar{n}$, and let us prove by induction on t that $n_0 + td$ is representable for all t. The base case $t = 0$ is the hypothesis that n_0 is representable.

For the inductive case, assume $n_0 + td$ is representable, and consider the non-emptying function$f_{(t+1)d}$. We get:

$$f_{(t+1)d}(\bar{n}) = \bar{n} + (t+1)d = \bar{n} + n_0 - \bar{n} + td = n_0 + td$$

that is representable by inductive hypothesis. By the instance of Lemma 4.7 for the pair $\{n_0, \bar{n}\}$ and the function $f_{(t+1)d}$, we get that $f_{(t+1)d}(n_0) = n_0 + (t+1)d$ is representable too, that is exactly the inductive step.

Since $\bar{n} \neq n_0$ also $d \neq 0$, hence $\{n_0 + td \mid t \in \mathbb{N}\}$ is infinite. Moreover $\{n_0 + td \mid t \in \mathbb{N}\} \subseteq R$ by the induction above, but this is impossible since R must be finite by Lemma 4.9. □

The meaning of this proposition for program analysis is the fact that a domain small enough (by Assumption 4.8) is probably unable to deduce meaningful information on an integer domain: if it does not contain representable singletons it must abstract to $\perp$ any variable initialization, and otherwise, it cannot be non-emptying for all sums, hence getting $\perp$ when values are manipulated using this operation. In both cases, because of strictness, the abstract $\perp$ is propagated along program paths, yielding it as the final result of the analysis, which means exactly it cannot determine any information. This issue is not bound to manifest for all programs, but for any domain there exist programs for which it does.

Note that the only hypothesis on the abstract domain is related to its size, by Assumption 4.8. This would include all classical numerical domains such as intervals, octagons or congruences [Gra91]. Of course, these are over-approximation domains, so this result does not apply to them, but it shows that our hypotheses are very general and would include many known abstract domains.

4.4.2 Finite integer domain

An analogous result can be obtained for a finite integer domain $\mathcal{P}([-N, N])$, where N is some big integer. This concrete domain models machine integers, that are constrained within an interval, so we assume that operations are performed in machine arithmetic, that is wrapping around in case of overflows. This is modelled working modulo $2N + 1$, the length of the interval, and taking the unique representative of each congruence class in the interval $[-N, N]$ of interest. It is worth noting that we take an interval that is symmetric around 0 to simplify notation, but there is no conceptual difference in using an asymmetric one instead.

Assumption 4.11. We assume that an abstract domain A, to be feasible for the concrete domain $\mathcal{P}([-N, N])$, must have a cardinality that is polynomial in N.

This assumption, just like Assumption 4.8, guarantees that for any set of input values the cost of the analysis is always polynomial in that of a single concrete execution, while the concrete analysis could require an exponential cost.

In the following, we use asymptotic notation for some quantities. Therefore, for each N we consider the concrete set of values $C_N = [-N, N]$ and define an abstract domains A_N for the concrete domain $\mathcal{P}([-N, N])$. Intuitively, each A_N represents the same abstraction, just instantiated for different values of N. With this notation, Assumption 4.11 becomes $|A_N| = O(\text{poly}(N))$.

The next lemma is analogous to Lemma 4.9 in proving that some sets are small under Assumption 4.11 on the cardinality of A_N. Here, the "exponential increase" we mentioned is explicit, as we are dealing with finite quantities. For brevity, we write R_N instead of R_{A_N}.

Lemma 4.12. *Let $S \subseteq [-N_0, N_0]$ for some N_0. Then, $|R_N(S)| = O(\log(N))$.*

The following proposition uses the same proof line as Proposition 4.10 above: we define a sequence of representable elements and prove that they are too many since, by the previous lemma, R_N is quite small.

Proposition 4.13. *Let $\langle \mathcal{P}([-N, N]) \overset{\alpha}{\leftrightarrows} A_N \rangle$ be a UGI for all N, and assume that there is an integer n_0 that is representable in all the A_N. Then there exists an N_0 such that, for all $N > N_0$, it cannot be the case that all the functions of the form $f_n(x) = x + n$ (modulo $2N + 1$) are non-emptying in A_N.*

Proof. Let $r_N = |R_N(n_0)|$. By the previous Lemma 4.12 we know that $r_N = O(\log(N))$. For all N, fix an element $\bar{n}_N \notin R_N(n_0)$ not representable with n_0 in A_N such that $d_N = n_0 - \bar{n}_N \leq r_N + 1$. This element exists because otherwise all elements in the interval $[n_0 - r_N - 1, n_0 - 1]$ (modulo $2N + 1$) would be representable with n_0, that is impossible since that interval contains $r_N + 1 = |R_N(n_0)| + 1$ elements.

Consider an N such that all the functions of the form $f_n(x) = x+n$ (modulo $2N+1$) are non-emptying in A_N. Following the proof of Proposition 4.10, we can show by induction on $t \geq 1$ that the value $f_{td_N}(\bar{n}) = n_0 + (t - 1)d_N$ is representable in A_N. All these values are different from one another for

$$1 \leq t < \frac{2N + 1}{d_N}$$

so that

$$|R_N| \geq \frac{2N + 1}{d_N}$$

However, we know that $|R_N| = O(\log(N))$ and $(2N + 1)/d_N = \omega(\log(N))$, therefore there exists some N_0 such that for any $N > N_0$ this inequality does not hold. This in turn implies that for all $N > N_0$ it is not possible that all the functions of the form $f_n(x) = x + n$ (modulo $2N + 1$) are non-emptying in A_N. □

Again, the only assumption on A_N is about its size, so our result applies to many kinds of domains. For instance, a predicate abstraction [GS97] with $\Omega(\log(N))$ predicates exceeds this bound, as its size is doubly exponential in the number n of predicates.

4.5 General infinite concrete domains

In this section, we try to extend the results of the previous Section 4.4.1 on the infinite concrete domain of integers to other infinite concrete domains. More precisely, we deal with an infinite set C of concrete values, and a UGI $\langle \mathcal{P}(C) \overset{\alpha}{\leftrightarrows} A \rangle$. Again, we take the Assumption 4.8 on the size of A. Under this assumption, we can prove again Lemma 4.9, that does not depend on the specific integer domain considered in the previous section.

All conditions we propose in this section are mainly on the family of functions considered and not on the abstract domain. The reason for this is that first we fix a function family, corresponding to a program, and then we look for a domain well suited to analyse the specific family at hand. In other words, the family is given by the applicative context, while the domain can be adapted to it.

Definition 4.14 (Highly surjective function family)**.** Given a family F of functions from C to itself and an element $c \in C$, let

$$P_F(c) = \{d \in C \mid \exists f \in F.\ f(d) = c\}$$

be the set of *preimages of* c, elements of C that can be mapped to c by a function in F. We say that the family F is *highly surjective* if $P_F(c)$ is infinite for any possible choice of $c \in C$.

This property is needed together with Lemma 4.9 to apply Lemma 4.7 and get a new representable element: since there are infinitely many preimages of c but $R(c)$ is finite, there are elements $\tilde{c} \in P_F(c)$ not in $R(c)$; then by definition of $P_F(c)$ there is an f such that $f(\tilde{c}) = c \in R$, so we can apply the lemma to get $f(c) \in R$. The reason for requiring $f(\tilde{c}) = c$ instead of just in R is that, at the beginning of the proof, we assume R to contain only one element, hence the two conditions are equivalent. Starting from this basic idea, we present two sets of sufficient conditions to prove the non existence of any under-approximation abstract domain.

4.5.1 Local requirements for impossibility

The first set of conditions we propose is in a sense more "local", in that it requires conditions on each function in the family F, independently of the others.

The proof of this result proceeds as follows: it starts from a representable $c_0 \in R$ and iteratively creates an infinite sequence $\{c_n\}_{n\in\mathbb{N}}$ of representable elements. This yields a contradiction since R should be finite by Lemma 4.9.

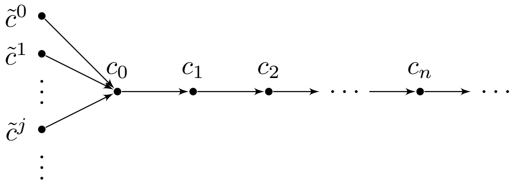


Figure 4.3: Graphical representation of the "final" f

The main idea is to pick a suitable f in F and define the sequence as the iterates $c_{n+1} = f(c_n)$. This function is sketched in Figure 4.3. The initial set of elements $\tilde{c}^j$ mapped to c_0 is required to apply Lemma 4.7 to all pairs $\{\tilde{c}^j, c_n\}$ and get that $c_{n+1} = f(c_n)$ is representable, since $f(\tilde{c}^j) = c_0 \in R$. The difficulty in realizing this idea is that we do not have enough information on the sequence to pick the right function f at the beginning, so we bring along a list of candidate functions that all coincide on a prefix of the sequence. At step n, we pick a new element of the sequence among the possible images of c_{n-1} through all candidate f we have at that point and discard all those functions that cannot match the choice. Actually, instead of directly considering functions, we represent them with elements $\tilde{c}$ of C. Each one represents a function $f_{\tilde{c}}$ that satisfies $f_{\tilde{c}}(\tilde{c}) = c_0$. Note that this can be done for "enough" (that is, infinitely many) $\tilde{c}$ because of the high surjectivity hypothesis. We call E_n the set of element $\tilde{c}$ that represent functions that are "valid" for the prefix up to n, i.e., they map c_i to c_{i+1} for $0 \leq i \leq n-1$. The core of the proof is an induction that proves that E_n always contains infinitely many elements and that the newly chosen c_n is different from all the previous ones.

Theorem 4.15. *Let F be a highly surjective function family from C to itself such that all functions $f \in F$ are either injective or acyclic. Assume also that $R \neq \emptyset$. Then there is at least one function $f \in F$ that is not non-emptying in A.*

Remark 4.16. In the previous section, we developed an ad hoc proof for the family of sums over integers in Proposition 4.10, but the same result can also be obtained as an application of Theorem 4.15: if $C = \mathbb{Z}$ and $F = \{\lambda x.x + n \,|\, n \in \mathbb{Z}\}$, the family is highly surjective (actually $P_F(c) = \mathbb{Z}$ for all c) and all these functions are injective, so it meets the hypotheses of the theorem. However, it is interesting to note that the proof of Theorem 4.15 is not a generalization of the proof of Proposition 4.10. Here we iterate a single f to build the entire sequence, while in the previous one we change the function every time, mapping the non representable $\tilde{n}$ to the newly found representable $n_0 + td$ to get that the image of n_0 through that function is representable too, as sketched in Figure 4.4.

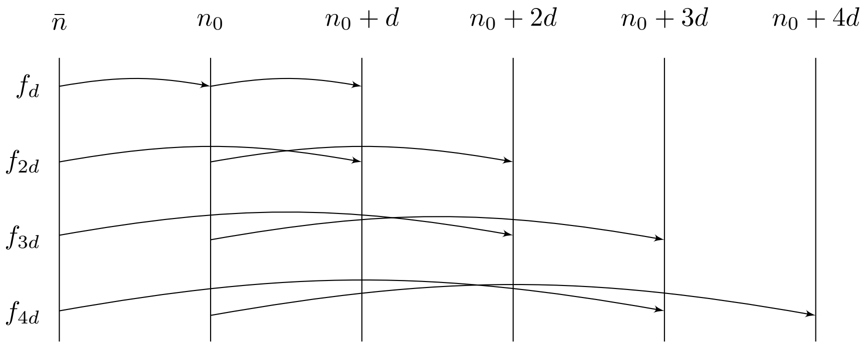


Figure 4.4: Graphical representation of the proof of Proposition 4.10

Another example are rational or real numbers, with sums or products.

Example 4.17. Take $C = \mathbb{Q} \setminus \{0\}$ and $F = \{\lambda x.x \cdot q \,|\, q \in \mathbb{Q} \setminus \{0\}\}$. The family is highly surjective since $P_F(c) = \mathbb{Q} \setminus \{0\}$ for all c, and all these functions are invertible, hence injective. ■

A possibly more interesting example of application are floating-point numbers as described by the IEEE Standard.

Example 4.18. Take $C = \mathcal{F} \setminus \{0\}$ the set of non-zero floating-point numbers that can be represented with a fixed number of significant digits, say t bits, but with an arbitrary precision exponent. We choose infinite precision exponents but a finite number of significant digits to have an infinite domain, as required by the theorem, but also to preserve characteristics of floating-point arithmetic.

Let $\cdot$ and $\odot$ denote respectively real product and its floating-point approximation, and consider the function family $F = \{\lambda x.x \odot y \,|\, y \in C\}$. The function family is highly surjective, e.g., considering that all numbers with the same significant digits as a floating-point x but different exponent can be mapped into x by multiplying them by 2 to the power of the difference of exponents. For the second condition, if $y = \pm 1$ we have that the function $\lambda x.x \odot y$ is invertible, hence injective. Otherwise, assume without loss of generality that $y > 1$ (other cases are analogous), and by contradiction assume it has a cycle $f^n(x_0) = x_0$. By monotonicity of $\odot$ we have $f(x) = x \odot y \geq x \odot 1 = x$, hence $x_0 \leq f(x_0) \leq f^2(x_0) \leq \cdots \leq f^n(x_0) = x_0$ so all the elements of the cycle are equal, and in particular $f(x_0) = x_0$. However, if $y \neq 1$, the product $x \odot y$ is never equal to x, that is a contradiction. Hence the function is acyclic. This means F meets the hypotheses of Theorem 4.15, hence no abstract domain on floating-point numbers can be non-emptying for all multiplications. ■

4.5.2 Global requirements for impossibility

The second set of conditions we propose is “global”, in the sense that it requires the family F to satisfy a property as a whole.

The proof of this theorem starts from the infinite set $P_F(c_0)$ and, using the hypotheses that some sets are finite, propagates its infiniteness down to R, yielding a contradiction with Lemma 4.9, stating that R is finite.

Theorem 4.19. *Let F be a highly surjective family of functions from C to itself such that*

1. *for all pair of elements $c, d \in C$, the set $\{f \in F \mid f(d) = c\}$ is finite;*

2. *for all pair of an element $c \in C$ and a function $f \in F$, the set $\{d \in C \mid f(d) = c\}$ is finite.*

Assume also that R is not empty. Then there is at least one function $f \in F$ that is not non-emptying in A.

Remark 4.20. Again, Theorem 4.19 can be used to prove Proposition 4.10, but the proofs are different. The former starts from the infinite set $P_F(c_0)\backslash R(c_0)$ of preimages $\tilde{c}$ of c_0 that are not representable with it. This yields an infinite list of pairs $\{\tilde{c}, c_0\}$ to apply Lemma 4.7: for any such pair, since $\tilde{c}$ is in $P_F(c_0)$, we get a function $f_{\tilde{c}}$ such that $f_{\tilde{c}}(\tilde{c}) = c_0 \in R$, so that $f_{\tilde{c}}(c_0) \in R$, too. The proof then exploits the remaining hypotheses to prove that there are infinitely many distinct $f_{\tilde{c}}(c_0)$. The proof of the latter relies on multiple functions to apply Lemma 4.7, but it always uses the same pair $\{\bar{n}, n_0\}$: at every step, it finds a function that maps $\bar{n}$ to the representable element found at the previous step.

Similarly to Theorem 4.15, this result can be used to prove the impossibility of building an abstract domain for floating-point numbers.

Example 4.21. Take $C = \mathcal{F} \setminus \{0\}$ the set of non-zero floating-point numbers with t bits significands and arbitrary precision exponents, and $F = \{\lambda x.x \odot z \,|\, z \in \mathcal{F} \setminus \{0\}\}$. As observed in Example 4.18 this family is highly surjective. Fixed now two floating-point numbers x, y, and letting $\mathbf{u}$ be the machine precision of floating-point arithmetic, we have that $y = x \odot z$ only if

$$\left|\frac{y - (x \cdot z)}{x \cdot z}\right| < \mathbf{u}$$

that is

$$\left|\frac{y}{x}\right| \frac{1}{1+\mathbf{u}} < |z| < \left|\frac{y}{x}\right| \frac{1}{1-\mathbf{u}}$$

This is a bounded interval since $x \neq 0$, and hence contains only a finite amount of floating-point numbers. This in turn means that, fixed x and y, there is only a finite amount of functions of the form $\lambda x.x \odot z$ such that $f(x) = y$. Analogously, fixed a floating-point y and a function $f(x) = x \odot z$, we have that $y = f(x)$ only if $|x|$ belong to a bounded interval, that contains a finite amount of floating-point numbers. So, fixed y and a function $f = \lambda x.x \odot z$, only a finite number of x satisfies $f(x) = y$. So, through Theorem 4.19 above, we proved again that no abstract domain on floating-point numbers can be non-emptying for all multiplications. ■

We point out once more that the only hypothesis on the abstract domain is about its size for both theorems in this section. For instance, our result applies to any under-approximation predicate abstraction domain [GS97]. Such a domain is defined by a (finite) list of predicates, each one representing the set of states satisfying the predicate, and an abstract state is a subset of predicates. By duality with respect to over-approximation, the concretization of such a set is the *union* of the states described by each predicate. Analogously, the abstraction of a set of concrete states S is the set of predicates which are *entirely contained* in S. Note that this interpretation is consistent with the use of under-approximations in Incorrectness Logic: the difference is that the logic can use any under-approximation formula, while predicate abstraction is constrained to use just a finite set of predicates, fixed beforehand.

4.5.3 On the necessity of high surjectivity

Both sets of conditions we proposed in this section require the function family to be highly surjective. This turns out to be necessary to prove that no under-approximation abstract domain exists, as we show in this section.

Proposition 4.22. *For any fixed family F of functions from C to itself that is not highly surjective, there exists an abstract domain A_F for $\mathcal{P}(C)$ such that:*

- *A_F is finite, and*
- *all functions $f \in F$ are non-emptying in A_F.*

Notably, the above proof is constructive. We present an example of such domain construction below.

Example 4.23. Fix the pair of functions $f(x) = x - 1$ and $g(x) = x - 2$ on $\mathbb{Z}$. The family $F = \{f, g\}$ is not highly surjective, so we build an under-approximation abstract domain for which these functions are non-emptying. First, take an integer n_0 such that $P_F(n_0)$ (computed with respect to F) is finite. With this F, any integer is a suitable candidate, so let us fix $n_0 = 0$.

The set of preimages of 0 is $P_F(0) = \{1, 2\}$. We define the abstract domain A_F as

$$A_F = \{\emptyset\} \cup \{X \cup \{0\} \mid X \subseteq P_F(0)\} = \{\emptyset, \{0\}, \{0,1\}, \{0,2\}, \{0,1,2\}\}.$$

In this abstract domain, a set is abstracted to $\emptyset$ if and only if it does not contain 0 since all elements of A_F but $\emptyset$ contains 0 and the abstraction of a set S must be a subset of S.

To check that f is non-emptying in A_F, fix a set $S \subseteq \mathbb{Z}$. If $\alpha(S) = \emptyset$ the non-emptying condition is vacuously true, so assume this is not the case, or, equivalently, that $0 \in S$. Analogously, if $\alpha(f(S)) = \emptyset$ the condition is true, so assume $0 \in f(S)$ or, equivalently, $1 \in S$. Using these two assumptions we get

$$\begin{aligned}
f^A(\alpha(S)) &= \alpha(f(\alpha(S))) && [\text{def. of } f^A]\\
&\supseteq \alpha(f(\alpha(\{0,1\}))) && [\alpha, f \text{ monotone}, S \supseteq \{0,1\}]\\
&= \alpha(f(\{0,1\})) && [\alpha(\{0,1\}) = \{0,1\}]\\
&= \alpha(\{-1,0\}) = \{0\} && [\text{def. of } f \text{ and } \alpha]
\end{aligned}$$

The check for g is analogous. ■ ■

Even though this proposition defines an under-approximation abstract domain, it should not be interpreted as a positive result since the resulting domain is almost a power set and hence too large to be feasible in practice. Instead, the proposition should be regarded as a way to show that one of the hypotheses required in the previous theorems is tight and cannot be weakened. Particularly, since these kinds of results require high surjectivity, they are ill-suited when the focus is on a single function.

This proposition can be generalized to consider sets $S \subseteq C$ whose preimages are finite, but a little care is needed when lifting the definition of preimages to sets of values: a preimage is a set for which there exists a function that maps it to S, not the union of the preimages of elements in S. Formally, we let:

$$P_F(S) = \{T \subseteq C \mid \exists f \in F.\ f(T) = S\}.$$

Using this definition, we can now easily generalize Proposition 4.22:

Proposition 4.24. *Let F be a family of functions from C in itself, and assume there is a set $S_0 \subseteq C$ such that $P_F(S_0)$ is finite. Then there exists a finite abstract domain A_F for $\mathcal{P}(C)$ such that all functions $f \in F$ are non-emptying in A_F.*

This proposition may also be applied to the concrete domain of finite lists to show that a natural function family to consider cannot be used to prove non existence of under-approximation domains using non-emptying functions.

Example 4.25. Fix the concrete domain C as the set of all lists of finite length over a finite, non-empty alphabet Γ, i.e. $C = \Gamma^*$. For $\alpha \in \Gamma^*$ a finite string, let

$$\mathrm{concat}_\alpha(\beta) = \alpha\beta$$

be the function that prefixes its argument by the string α. The family

$$F = \{\mathrm{concat}_\alpha \,|\, \alpha \in \Gamma^*\}$$

is not highly surjective, because fixed a string γ only its suffixes can be mapped into γ by a function in F, and they are a finite amount. Hence we can define an under-approximation abstract domain for which all these functions are non-emptying by means of Proposition 4.24. Such domains are defined with a construction similar to that of Example 4.23, and in particular, if ϵ is the empty list, considering the set $S_0 = \{\epsilon\}$ whose preimage is only S_0 itself, the construction yields

$$A_F = \{\emptyset, \{\epsilon\}\}.$$

It is easy to check that all functions concat_α are non-emptying in this abstract domain. ■

The previous proposition focuses on preimages, stating that if there is a concrete element that has a finite amount of preimages then it is possible to define an under-approximation domain. A natural dual of this proposition can be formulated in terms of images. For a subset $S \subseteq C$, the set of its images is defined as follows:

$$I_F(S) = \{f(S) \,|\, f \in F\}.$$

This definition is exactly dual to that of preimages and can be used to formulate a similar result.

Proposition 4.26. *Let F be a family of total functions (ie. if $S \neq \emptyset$ then $f(S) \neq \emptyset$) from $\mathcal{P}(C)$ in itself, and assume there is a non-empty set $S_0 \subseteq C$ such that $I_F(S_0)$ is finite. Then there exists a finite abstract domain A_F such that all functions $f \in F$ are non-emptying in A_F.*

Even though Proposition 4.26 introduces the technical hypothesis that all $f \in F$ are total, this condition is not very restrictive in practice, because our results are applicable when F is a family of basic transfer functions, that seldom introduce divergence: in programming languages, non-termination is often due to control-flow constructs and not to single assignments or guards. As an immediate application of the proposition that exploits images instead of pre-images, we consider lists again, and rule out another natural function family.

Example 4.27. Fix again $C = \Gamma^*$, and consider all functions of the form $\mathrm{drop}_n : \Gamma^* \to \Gamma^*$ that, taken a list, drop its first n elements and return the resulting list. If the input list is shorter than n, the output of drop_n is the empty list ϵ. The function family

$$F = \{\mathrm{drop}_n \,|\, n \in \mathbb{N}\}$$

is highly surjective since, for any fixed list $\alpha \in \Gamma^*$ and any n, we can prefix α by n arbitrary characters, and map this extended list back to α via drop_n. However, images through this function family are finite:

$$I_F(\alpha) = \{\text{drop}_n(\alpha) \mid n \in \mathbb{N}\}$$

is finite because $I_F(\alpha)$ coincides with the set of all tails of α. By Proposition 4.26 we can define an under-approximation abstract domain such that all functions drop_n are non-emptying. Again, these domains are constructed from sets S_0 with a finite amount of images, and considering $S_0 = \{\epsilon\}$, that satisfies $I_F(S_0) = \{\epsilon\}$, we get

$$A_F = \{\emptyset, \{\epsilon\}\}.$$

It can be easily checked that all functions drop_n are non-emptying in A_F. ■

These last two propositions consider opposite situations in which it is possible to define an under-approximation domain: the former requires to be able to go backward using F in infinitely many ways, while the latter to go forward. This often is not the case in the presence of "boundaries" in the concrete domain, which are points with respect to which functions tend to walk either up or away: for instance, ϵ is such a point with finite strings because concat functions go away from it while drop functions move towards it. Another example of such a boundary is 0 in the domain of integers $\mathbb{Z}$ for multiplications and (rounded) divisions: the former increase absolute value, moving away from 0 (even though 0 itself is never a preimage), while the latter decrease it. Also considering a function family made of both kinds of functions does not work: a slight adaptation of the constructions for the two propositions above shows that, if F can be partitioned into two subfamilies, each satisfying the hypothesis of one of the two propositions, then there exists an under-approximation abstract domain. An example of this is in the set of finite lists, taking as F both concat and drop functions. The construction then yields exactly $A_F = \{\emptyset, \{\epsilon\}\}$, for which all these functions are non-emptying, as shown in Examples 4.25 and 4.27. In light of these observations, to apply the definition of non-emptying function in an effective way for proving the non existence of abstract domains, for all possible boundaries there is the need for a function that can both enter and exit it. This happens for integers since there is no boundary, but does not for finite lists, with $\{\epsilon\}$ being often either a sink or a source for many functions on lists.

4.6 General finite concrete domains

The discussion of Section 4.5 requires C to be infinite. While this is a common simplification, since concrete domains have usually a very big size, an interesting and important question is whether our findings can be extended to the case of finite concrete domains. However, we believe that the two Theorems 4.15 and 4.19 cannot be straightforwardly adapted to the finite setting.

For both results, the proof showed the existence of infinitely many representable elements, contradicting Lemma 4.9. For a finite C, if $N = |C|$, one possibility would be to resort to Lemma 4.12 to show that $|R| = O(\log(N))$ and get the contradiction by proving the existence of $\omega(\log(N))$ representable elements. Unfortunately, similar constructions do not yield the desired bound: when C is infinite we exploited the fact that finite combinations of finite numbers are also finite, while for finite C we should require that arbitrary combinations of logarithmic factors are $O(N)$, which is not true.

The definition of highly surjective family itself is not easy to translate. The construction of Proposition 4.22 on a finite C yields an abstract domain with the needed features already when $|P_F(c)| = O(\log(N))$. However, to carry out proofs along the lines of Theorems 4.15 and 4.19 we would need stronger hypothesis than just $|P_F(c)| = \omega(\log(N))$, possibly up to $|P_F(c)| = \Theta(N)$.

Even with these theoretical considerations against finite counterparts of Theorems 4.15 and 4.19, we have been able to carry out the proof for the special case of the finite domain $C = [-N, N]$ of integers (Proposition 4.13). Our result exploits the precise structure of both the given concrete domain and function family, particularly two key points. First, functions produce elements that are not "too far away" with respect to the size of the domain (i.e., $n_{i+1} - n_i = O(\log(N))$), and this allows to prove that there are enough distinct representable elements. Second, the domain is circular and hence has no boundaries. If the domain had them, we could have applied results such as Example 4.25 or Proposition 4.26. This was not the case for integers because additions overflow, so the domain has no boundaries near $-N$ and N.

Building on the above discussion, we present a version of the global condition for finite concrete domains. To overcome the limitation of boundaries, we explicitly constrain the initial representable element c_0, writing hypotheses around it. Such hypotheses imply that c_0 is "far enough" from the boundaries of the domain (if any). Conditions (1-2) of the next theorem correspond to those of the infinite version (Theorem 4.19) rewritten with c_0 in mind. The relation between the two bound functions k_1 and k_2 and the number of preimages of c_0 corresponds to the high surjectivity hypothesis. It only constrains the value c_0 because this is the representable value from which the proof begins.

As we did in the specific case of the integer domain $[-N; N]$, we assume the size of A to be polynomial in N (Assumption 4.11). To say this formally, as we did for finite integers (cfr. Section 4.4.2), we consider a sequence C_N of sets of concrete values with size $O(N)$. In the specific instance of integers, such sets were the intervals $[-N; N]$; in general, we require that $C_{N-1} \subseteq C_N$ for all N. This inclusion formalizes the intuition that all the C_N are the "same" concrete domain instantiated for different sizes. We also assume to have an abstract domain A_N for each concrete domain $\mathcal{P}(C_N)$ such that $|A_N| = O(\mathrm{poly}(N))$. As before, we write R_N for R_{A_N}, and we remark that Lemma 4.12 holds.

Theorem 4.28. *Assume $|C_N| = O(N)$, and let $\langle \mathcal{P}(C_N) \overset{\alpha_N}{\leftrightarrows} A_N \rangle$ be a UGI for all N. Assume there exists a number N_1 and a value $c_0 \in C_{N_1}$ such that, for all $N > N_1$, c_0 is representable in A_N (i.e., $c_0 \in R_N$). Given two functions $k_1, k_2 : \mathbb{N} \to \mathbb{N}_{>0}$, for any N let F_N be a family of functions from C_N to itself such that:*

1. *for all elements $d \in C_N$, $|\{f \in F_N \mid f(c_0) = d\}| \leq k_1(N)$;*
2. *for all functions $f \in F_N$, $|\{d \in C_N \mid f(d) = c_0\}| \leq k_2(N)$.*

Lastly, assume that $|P_{F_N}(c_0)| = \omega(\log(N) \cdot k_1(N) \cdot k_2(N))$.
Then, there exists N_0 such that, for all $N > N_0$, it is not possible that all $f \in F_N$ are non-emptying in A_N.

A straightforward corollary is that, whenever we can verify the hypotheses for all values in C, there is no under-approximation domain with a representable element. This is for instance the case for integers and sums, so we recover Proposition 4.13:

Example 4.29. Let $C = [-N, N]$ and

$$F = \{\lambda x.x + n(\text{modulo } 2N + 1) \mid n \in [-N, N]\}.$$

Fixed any $n_0 \in [-N, N]$, it is easy to check that $P_F(n_0) = [-N, N]$ and $k_1(N) = k_2(N) = 1$. The condition

$$|P_F(n_0)| = \omega(\log(2N + 1) \cdot k_1(2N + 1) \cdot k_2(2N + 1))$$

thus reduces to

$$2N + 1 = \omega(\log(N))$$

that is true. Hence there is no under-approximation abstract domain with at least one integer n_0 representable. ■

The example of integers has the nice property of not having boundaries, but this is seldom the case. For instance, consider floating point numbers: they overflow and underflow to special values, hence they do have boundaries. Nevertheless, if we pick a suitable initial element c_0 we can still apply the theorem above.

Example 4.30. Consider the finite set of non-zero floating-point numbers $C = \mathcal{F} \setminus \{0\}$ with t bits significand, one bit sign and e bit exponents. Consider the function family $F = \{\lambda x.x \odot z \,|\, z \in \mathcal{F} \setminus \{0\}\}$ of floating-point multiplications, where $\odot$ denotes the floating-point approximation of real product.

As shown in Example 4.21, fixed two floating-point numbers x, y, we have that $y = f(x) = x \odot z$ only if

$$|z| \in \left[\left|\frac{y}{x}\right| \frac{1}{1+\mathbf{u}}, \left|\frac{y}{x}\right| \frac{1}{1-\mathbf{u}}\right].$$

If $\mathbf{u} \leq 1/2$ this is entirely contained in the interval

$$\left[\left|\frac{y}{x}\right|(1-2\mathbf{u}), \left|\frac{y}{x}\right|(1+2\mathbf{u})\right].$$

It can be shown that, if $\mathbf{u} < 1/2$, the quantity of floating point numbers in that interval is bounded by a constant c that does not depend on x and y.[1] Analogously, fixed y and z there are at most c floating point numbers x such that $y = f(x) = x \odot z$.

With these two bounds, if x_0 has enough preimages we can verify the last hypothesis of the theorem: letting $N = |C|$

$$|P_F(x_0)| = \omega(\log(N) \cdot k_1(N) \cdot k_2(N)) = \omega(\log(N) \cdot c \cdot c) = \omega(\log(N)).$$

For instance $x_0 = 1$ satisfies this condition (more than half of all floating-point numbers have a floating-point inverse, hence $|P_F(x_0)| \geq N/2 = \Theta(N)$) but there are many other points satisfying it. By mean of Theorem 4.28, no under-approximation abstract domain is non-emptying for all multiplications whenever one of these points is representable. ■

4.7 Conclusions

In this chapter, we pointed out some asymmetries between over and under-approximation in Abstract Interpretation, and why those are an obstacle to the design of abstract domains for program analysis via Under-approximation Galois connections. The key observation is that the duality between under and over-approximation is broken by the fact that in program analysis over and under-approximations have to be applied to the same transfer functions. The handling of divergence in the abstract domain poses another critical issue.

Building on those ideas, we proposed the novel definition of *non-emptying function* and studied how it plays a crucial role in proving the non-existence of general, useful under-approximation based abstract domains. Indeed, our results prove that an analysis based on such domains will very often answer $\bot$ (representing either absence of information or divergence) for programs that require repeated applications of non-emptying functions. This is a big limitation since recovery from $\bot$ in an under-approximation is not as easy as

[1] For instance, this can be proved for $c = 17$.

recovering from $\top$ in an over-approximation: we can say that it is quite impossible. We applied our general results to several concrete domains to conclude that, under some mild assumptions, there are no useful under-approximation abstract domains. Then, to show that one of the hypotheses in our result is tight, we proposed a construction to craft an under-approximation abstract domain whenever such a hypothesis is not met.

To summarize, our results hint at the difficulty of designing under-approximation abstract domains. This suggest that it's better to resort to other under-approximation techniques to combine with over-approximation. Therefore, in the next chapter we study logical frameworks for under-approximation.

Chapter 5

A comparison of program logics

In this chapter, we consider three known triple-based program logics, namely Hoare Logic (HL), Incorrectness Logic (IL) and Necessary Conditions (NC). We characterize their validity conditions in term of over or under-approximation of forward and backward semantics. First, this allows us to identify the absence of one combination, and thus to define a new program logic, called Sufficient Incorrectness Logic (SIL), for it. Second, this guides a thorough comparison of the four validity conditions, highlighting analogies and differences between over and under-approximation approaches.

The content of this chapter is based on [ABGL24].

5.1 Taxonomy

As discussed in Sections 2.4 and 2.5, the validity conditions of HL and IL are defined as over and under-approximation of the forward semantics $\llbracket \cdot \rrbracket$. However, other program logics cannot be naturally described in terms of $\llbracket \cdot \rrbracket$. To this end, we consider a backward semantics $\llbracket \overleftarrow{\cdot} \rrbracket$ defined as the converse relation of the forward semantics,[1] that is

$$\llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma' \triangleq \{ \sigma \mid \sigma' \in \llbracket \mathsf{r} \rrbracket \sigma \} \tag{5.1}$$

or, equivalently,

$$\sigma \in \llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma' \iff \sigma' \in \llbracket \mathsf{r} \rrbracket \sigma \tag{5.2}$$

and we additively lift this definition to set of states by union. Intuitively, the forward semantics $\llbracket \mathsf{r} \rrbracket P$ denotes the set of all possible output states of r when execution starts from a state in P (and r terminates). Instead, the backward semantics $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$ denotes the set of all input states that can lead to a state in Q.

The backward semantics can also be characterized compositionally, similarly to the forward one:

Lemma 5.1. *For any regular commands* $\mathsf{r}, \mathsf{r}_1, \mathsf{r}_2 \in \mathsf{Reg}$*, the following equalities hold:*

$$\llbracket \overleftarrow{\mathsf{r}_1 ; \mathsf{r}_2} \rrbracket = \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \circ \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \qquad \llbracket \overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2} \rrbracket = \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \cup \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \qquad \llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket = \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n$$

Using $\llbracket \overleftarrow{\cdot} \rrbracket$, we characterize NC with the validity condition $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \subseteq P$. To see why, assume Q describes good final states. Then, $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$ defines all states which can reach a

[1] Formally, if we consider a relation $\mathcal{R}$ between states defined as $\sigma \mathcal{R} \sigma'$ iff $\sigma' \in \llbracket \mathsf{r} \rrbracket \sigma$, the backward semantics $\llbracket \overleftarrow{\mathsf{r}} \rrbracket$ defines the converse relation $\mathcal{R}^{-1}$

	Forward		Backward
Over	HL: $[\![\mathsf{r}]\!]P \subseteq Q$	$\xleftrightarrow{\simeq}$	NC: $[\![\overleftarrow{\mathsf{r}}]\!]Q \subseteq P$
Under	IL: $[\![\mathsf{r}]\!]P \supseteq Q$		SIL: $[\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P$

Figure 5.1: The taxonomy of validity conditions. Columns indicate if validity is based on forward or backward semantics and rows the verse of approximation. HL and NC are equivalent ($\simeq$). SIL is our new logic.

good state. Since a precondition $\underline{P}$ is necessary for Q if it contains every state which can reach a state in Q, $\underline{P}$ must contain at least all states in $[\![\overleftarrow{\mathsf{r}}]\!]Q$. More formally, for any initial state $\sigma \in [\![\overleftarrow{\mathsf{r}}]\!]Q$, there exist a $\sigma' \in [\![\mathsf{r}]\!]\sigma$ such that $\sigma' \in Q$. This means that σ has a trace in $\mathcal{T}(\sigma)$ (the one ending in σ'), so it must belong to any necessary precondition $\underline{P}$.

Proposition 5.2 (NC as backward over-approximation)**.** *Given a postcondition Q for the program* r*, any possible necessary precondition $\underline{P}$ for Q satisfies*

$$[\![\overleftarrow{\mathsf{r}}]\!]Q \subseteq \underline{P}$$

Note that this is not a new understanding of NC (it was already hinted in the work that introduced them, and an analogous characterization appeared in [ZK22, §6.3] in terms of weakest precondition), but the explicit use of the backward semantics in our contest enables a more streamlined comparison with other logics in Section 5.3.

We organize the validity conditions of HL, IL and NC in the taxonomy in Figure 5.1. We classify logics depending on (1) whether the condition is expressed in terms of forward or backward semantics and (2) whether it is an over or an under-approximation. This naturally sparks the question on what does the backward under-approximation condition mean and whether a logic for it has been developed.

Backward under-approximation and Lisbon Triples At POPL'19 in Lisbon, D. Dreyer and R. Jung suggested that P. O'Hearn should look at bug-finding in terms of a logic for proving the presence of faults (as reported in [OHe20; ZDS23]). However, the proposed model of triples did not fit well with a key feature of Pulse, a bug-catching tool developed at Meta, namely its ability to drop the analysis of some program paths, for which IL provides a sound logical foundation instead. The idea of such "Lisbon" triples is that *for any initial state satisfying the pre, there exists some execution trace leading to a final state satisfying the post* and it can be dated back to Hoare's calculus of possible correctness [Hoa78], even if no form of approximation was considered there. Lisbon triples were then briefly discussed in [MOH21, §5] and [Le+22, §3.2] under the name *backwards under-approximate triples*. They were also one of the motivations for OL [ZDS23]: OL recognized the importance of tracking the sources of errors and can encode Lisbon triples together with Hoare triples. Backward under-approximation was also key for the development of a compositional non-termination analysis that has been integrated in Pulse [RVO24], based on the observation that forward and backward under-approximation can be unified in a single logical framework by dropping the respective consequence rules.

Ideally, given an incorrectness specification, the goal of backward under-approximation would be to report to programmers all dangerous input states that lead to bugs. However, all the above proposals are designed according to the forward semantics of programs and thus are best suited to infer postconditions starting from some given precondition. To

Logic	HL [Hoa69]	IL [OHe20]	NC [CCL11]	OL [ZDS23]	SIL [ABGL24]
Triples	$\{P\}\ \mathsf{r}\ \{Q\}$	$[P]\ \mathsf{r}\ [Q]$	$(P)\ \mathsf{r}\ (Q)$	$\langle P\rangle\ \mathsf{r}\ \langle Q\rangle$	$\langle\!\langle P\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle$

Figure 5.2: Summary of the notation for different program logics

tackle this issue, we introduce Sufficient Incorrectness Logic (SIL) as a proof system for Lisbon triples with backward-oriented rules.

Since in this chapter we deal with different kinds of program logics, a summary of the notation for the various triples is reported in Figure 5.2.

5.2 Sufficient Incorrectness Logic

SIL is a backward-oriented under-approximation proof system for Lisbon triples that focuses on *finding the sources of incorrectness rather than highlighting the presence of bugs.* Assuming the post Q defines some class of errors, i.e., it is what we call an *incorrectness specification*, a (valid) SIL triple $\langle\!\langle P\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle$ means that "*all input states that satisfy P have at least one execution of the program* r *leading to a state that satisfies Q*". For deterministic programs, SIL guarantees that a state satisfying the pre *always* leads to an error. Instead, if r is nondeterministic, SIL guarantees that there *exists* an execution that leads to an error. Sufficient incorrectness preconditions are extremely valuable to programmers: by pointing out the sources of errors, they serve as a starting point to scope down debugging, fuzzing, and testing. Moreover, it is known that, contrary to IL and its extensions, backward under-approximation can expose manifest errors [Le+22, §3.2]: an error Q is manifest iff the SIL triple $\langle\!\langle \mathbf{true}\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle$ is valid. These are bugs that happen regardless of the context and it has been observed experimentally that are more likely to be fixed when reported [Le+22, §5]. Therefore, we study SIL to enhance program analysis frameworks with the ability to identify the source of incorrectness.

SIL goals include: (i) defining a default deduction mechanism that starts from a specification of erroneous outcomes and traces the computation back to some initial states responsible for such errors; (ii) exhibiting a minimal set of rules that are sound and complete for Lisbon triples; and (iii) spelling out, a posteriori, a formalization of the backward analysis step performed by industrial grade analysis tools for security developed at Meta [DFLO19; Gab21; BC19]. Those tools automatically find more than 50% of the security bugs in the Meta family of apps and many severe bugs [DFLO19, Fig. 5].

Roughly, the SIL triple

$$\langle\!\langle P\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle$$

requires that all states in P have at least one execution leading to a state in Q. More formally, for all $\sigma \in P$ there must exist a state $\sigma' \in Q$ such that $\sigma' \in [\![\mathsf{r}]\!]\sigma$. Particularly, in the presence of nondeterminism states in P are required to have one execution leading to Q, not necessarily all of them. Motivated by Figure 5.1, we choose the validity condition

$$[\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P. \tag{SIL}$$

However, the two definition are equivalent:

Proposition 5.3 (Characterization of SIL validity)**.** *For any* $\mathsf{r} \in \mathsf{Reg}$*,* $P, Q \subseteq \Sigma$

$$[\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P \iff \forall \sigma \in P \,.\, \exists \sigma' \in Q \,.\, \sigma' \in [\![\mathsf{r}]\!]\sigma$$

$$\frac{}{\langle\!\langle [\![\overleftarrow{\mathsf{c}}]\!]Q \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{atom}\rangle\!\rangle \qquad \frac{P \subseteq P' \quad \langle\!\langle P' \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q' \rangle\!\rangle \quad Q' \subseteq Q}{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{cons}\rangle\!\rangle$$

$$\frac{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle R \rangle\!\rangle \quad \langle\!\langle R \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}_1; \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{seq}\rangle\!\rangle \qquad \frac{\langle\!\langle P_1 \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle Q \rangle\!\rangle \quad \langle\!\langle P_2 \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P_1 \cup P_2 \rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{choice}\rangle\!\rangle$$

$$\frac{\forall n \geq 0 \,.\, \langle\!\langle Q_{n+1} \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_n \rangle\!\rangle}{\langle\!\langle \bigcup_{n\geq 0} Q_n \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q_0 \rangle\!\rangle}\ \langle\!\langle\mathsf{iter}\rangle\!\rangle$$

Additional rules

$$\frac{}{\langle\!\langle \emptyset \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{empty}\rangle\!\rangle \qquad \frac{}{\langle\!\langle Q \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{iter0}\rangle\!\rangle$$

$$\frac{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}^\star; \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle\mathsf{unroll}\rangle\!\rangle \qquad \frac{\langle\!\langle P_1 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_1 \rangle\!\rangle \quad \langle\!\langle P_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_2 \rangle\!\rangle}{\langle\!\langle P_1 \cup P_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_1 \cup Q_2 \rangle\!\rangle}\ \langle\!\langle\mathsf{disj}\rangle\!\rangle$$

Figure 5.3: Sufficient Incorrectness Logic

A convenient way to exploit SIL is to assume that the analysis takes as input the incorrectness specification Q, i.e., the set of erroneous final states. Then, any valid SIL triple $\langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$ yields a precondition which surely captures erroneous executions. In this sense, P gives a sufficient condition for incorrectness and motivates the name of SIL. This is dual to the interpretation of IL where, for a given precondition P, any IL triple $[P]\ \mathsf{r}\ [Q]$ yields a set Q of final states which are for sure reachable, so that any error state in Q is a true bug reachable from some input in P.

5.2.1 Proof system

The inference rules for SIL are in Figure 5.3. The top five rules form a minimal, sound and complete proof system. The additional rules are other valid rules that can ease program analysis and are discussed in Section 5.2.2. Note that all the rules can be applied to an arbitrary post Q to infer a corresponding pre, in the same way as all the rules of IL can be applied to an arbitrary pre P to infer a corresponding post. This is a key feature of SIL proof system, which facilitates backward reasoning.

Atomic commands are handled by $\langle\!\langle\mathsf{atom}\rangle\!\rangle$, which exploits the backward semantics and summarizes cases for `skip`, assignments and Boolean guards. In Section 5.3.2 we discuss further the rule for assignment when pre and postconditions are formulae instead of sets of states. The consequence rule allows to generalize a proof by weakening/strengthening the two conditions P and Q involved. It can readily be derived from the validity condition (SIL). Moreover, $\langle\!\langle\mathsf{cons}\rangle\!\rangle$ allows SIL to drop disjuncts in the pre, just as $[\mathsf{cons}]$ allows IL do it in the post. This feature is crucial in both SIL and IL to increase scalability of tools. Rule $\langle\!\langle\mathsf{seq}\rangle\!\rangle$ is standard: SIL triples are composed sequentially just like in all other logics. Rule $\langle\!\langle\mathsf{choice}\rangle\!\rangle$ states that if all states in P_1 (resp. P_2) have an execution of r_1 (resp. r_2) ending in Q, they also have an execution of $\mathsf{r}_1 \oplus \mathsf{r}_2$ ending in Q since the semantics of $\mathsf{r}_1 \oplus \mathsf{r}_2$ is a superset of that of r_1 (resp. r_2), cf. Figure 2.1. This rule is also reminiscent of the equation for conditionals in the calculus of possible correctness [Hoa78]. For iteration, for each $n \geq 0$ we find inductively the precondition Q_n of executing r exactly n times. The precondition of the whole $\mathsf{r}^\star$ is then the union of all the Q_n, as formalized by rule $\langle\!\langle\mathsf{iter}\rangle\!\rangle$, which first appeared in [MOH21, §5].

The SIL proof system is both correct and complete. Correctness can be proved by induction on the derivation tree of a triple. Intuitively, if the premises of a rule are valid, then its consequence is valid as well, as we briefly observed in above. To prove completeness, we rely on the fact that rules other than $\langle\!\langle \mathsf{cons} \rangle\!\rangle$ are exact, that is, if their premises satisfy the equality $[\![\overleftarrow{\mathsf{r}}]\!]Q = P$, their conclusion does as well. Using this, we prove the triple $\langle\!\langle [\![\overleftarrow{\mathsf{r}}]\!]Q \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle Q \rangle\!\rangle$ for any r and Q. We conclude using $\langle\!\langle \mathsf{cons} \rangle\!\rangle$ to get a proof of $\langle\!\langle P \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle Q \rangle\!\rangle$ for any $P \subseteq [\![\overleftarrow{\mathsf{r}}]\!]Q$.

Theorem 5.4 (SIL is sound and complete)**.** *A SIL triple is provable iff it is valid:*

$$\vdash \langle\!\langle P \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle Q \rangle\!\rangle \iff \models \langle\!\langle P \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle Q \rangle\!\rangle$$

5.2.2 Additional rules for program analysis

The topmost set of five rules in Figure 5.3 is deliberately minimal: if we remove any rule it is no longer complete. However, there are other valid rules, not derivable from those five, that can be useful in practice. Some of them are at the bottom of Figure 5.3.

Rule $\langle\!\langle \mathsf{empty} \rangle\!\rangle$ is used to drop paths backward, just like IL can drop them forward (an analogous axiom $[P] \ \mathsf{r} \ [\emptyset]$ is valid for IL). Particularly, this allows to ignore one of the branches of $\langle\!\langle \mathsf{choice} \rangle\!\rangle$, or to stop the backward iteration of $\langle\!\langle \mathsf{iter} \rangle\!\rangle$ without covering all the iterations. An example of such an application is the derived rule $\langle\!\langle \mathsf{iter0} \rangle\!\rangle$, which corresponds to not entering the iteration at all. It can be derived from rules $\langle\!\langle \mathsf{iter} \rangle\!\rangle$ and $\langle\!\langle \mathsf{empty} \rangle\!\rangle$ by taking $Q_0 = Q$ and $Q_n = \emptyset$ for $n \geq 1$. It subsumes HL's rule $\{\mathsf{iter}\}$, which is based on loop invariants: those are a correct but not complete reasoning tool for under-approximation [OHe20]. Rule $\langle\!\langle \mathsf{unroll} \rangle\!\rangle$ allows to unroll a loop once. Subsequent applications of this rule allow to simulate (backward) a finite number of iterations, and then rule $\langle\!\langle \mathsf{iter0} \rangle\!\rangle$ can be used to ignore the remaining ones. This is on par with IL ability to unroll a loop a finite number of times to find some post, for which analogous rules are valid [OHe20; MOH21]. Rule $\langle\!\langle \mathsf{disj} \rangle\!\rangle$ allows to split the analysis and join the results, just like HL and IL. However, while a corresponding rule $\{\mathsf{conj}\}$ which perform intersection is sound for HL, it is unsound for both IL and SIL. We discuss this point further in Section 5.3.3.

All four these additional rules are sound:

Proposition 5.5 (Soundness of additional SIL rules)**.** *The additional SIL rules at the bottom of Figure 5.3 are sound, that is, triples provable in SIL extended with those rules are valid.*

Example 5.6. Let us consider the program "loop0" from [OHe20, §6.1]:

```
x := 0;
n := nondet();
while(n > 0) {
    x := x + n;
    n := nondet();
}
// assert(x != 2000000)
```

We can translate it in the syntax of regular commands by letting

$$\mathsf{r}_w \triangleq \texttt{(n > 0)?; x := x + n; n := nondet()}$$

$$\mathsf{rloop0} \triangleq \texttt{x := 0; n := nondet(); } (\mathsf{r}_w)^\star\texttt{; (n <= 0)?}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\!\langle T_{2M}\rangle\!\rangle\ \mathsf{r}_w^\star\ \langle\!\langle T_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{iter0}\rangle\!\rangle \quad \dfrac{\vdots}{\langle\!\langle T_{2M}\rangle\!\rangle\ \mathsf{r}_w\ \langle\!\langle R_{2M}\rangle\!\rangle}}{\langle\!\langle T_{2M}\rangle\!\rangle\ \mathsf{r}_w^\star;\mathsf{r}_w\ \langle\!\langle R_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{seq}\rangle\!\rangle}{\langle\!\langle T_{2M}\rangle\!\rangle\ \mathsf{r}_w^\star\ \langle\!\langle R_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{unroll}\rangle\!\rangle \quad \dfrac{}{\langle\!\langle R_{2M}\rangle\!\rangle\ \texttt{(n <= 0)?}\ \langle\!\langle Q_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{atom}\rangle\!\rangle}{\dfrac{\langle\!\langle T_{2M}\rangle\!\rangle\ \mathsf{r}_w^\star;\texttt{(n <= 0)?}\ \langle\!\langle Q_{2M}\rangle\!\rangle}{(*)}}\,\langle\!\langle\mathsf{seq}\rangle\!\rangle$$

$$\dfrac{\dfrac{\dfrac{}{\langle\!\langle\mathbf{true}\rangle\!\rangle\ \texttt{x := 0}\ \langle\!\langle x\le 2000000\rangle\!\rangle}\,\langle\!\langle\mathsf{atom}\rangle\!\rangle \quad \dfrac{}{\langle\!\langle x\le 2000000\rangle\!\rangle\ \texttt{n := nondet()}\ \langle\!\langle T_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{atom}\rangle\!\rangle}{\langle\!\langle\mathbf{true}\rangle\!\rangle\ \texttt{x := 0; n := nondet()}\ \langle\!\langle T_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{seq}\rangle\!\rangle \quad (*)}{\langle\!\langle\mathbf{true}\rangle\!\rangle\ \mathsf{rloop0}\ \langle\!\langle Q_{2M}\rangle\!\rangle}\,\langle\!\langle\mathsf{seq}\rangle\!\rangle$$

Figure 5.4: Derivation of the SIL triple $\langle\!\langle\mathbf{true}\rangle\!\rangle$ rloop0 $\langle\!\langle Q_{2M}\rangle\!\rangle$ for Example 5.6.

Final error states are those in $Q_{2M} \triangleq (x = 2000000)$.

To prove a triple for rloop0, we have to perform at least one iteration, and we do so using ⟪unroll⟫. We let $R_{2M} \triangleq (x = 2000000 \land n \le 0)$ and $T_{2M} \triangleq (x+n = 2000000 \land n > 0)$. It is straightforward to prove $\langle\!\langle T_{2M}\rangle\!\rangle$ r_w $\langle\!\langle R_{2M}\rangle\!\rangle$ via ⟪seq⟫ and ⟪atom⟫. Given this triple, we can unroll the loop once and prove the same triple for $\mathsf{r}_w^\star$, as shown by the combination of ⟪unroll⟫, ⟪seq⟫ and ⟪iter0⟫ at the top-left of Figure 5.4. This is a property of under-approximation: a nondeterministic number of iterations can be under-approximated by a single iteration [OHe20, §6.1]. With this triple for the loop, we can prove for the whole program the triple $\langle\!\langle\mathbf{true}\rangle\!\rangle$ rloop0 $\langle\!\langle Q_{2M}\rangle\!\rangle$ using ⟪seq⟫ and ⟪atom⟫. The full derivation is in Figure 5.4.

Differently than IL, this triple highlights that any initial state can lead to an error: instead of reporting the presence of a true bug, we can prove that this is a manifest error and produce an initial state which causes the bug. ■

5.3 Relations among logics

We first follow the two-dimensional scheme in Figure 5.1 to carry out a duality-driven comparison among the four validity conditions. Then, we realize that analogies and differences between them can be studied along other axes to obtain interesting insights among the relations between over/under-approximation, forward/backward analysis, reachability/divergence, and others.

We named columns of Figure 5.1 based on which semantics (forward or backward) they use. However, there is not a unique way of fixing the over and under-approximation axes. For instance, it is possible to take the consequence rules as the approximation axis, naming IL and NC as under-approximation because you can always substitute Q for one of its under-approximations $Q' \subseteq Q$. However, we chose to denote approximation depending on the "target" set, that is, Q for forward and P for backward semantics, respectively. We motivate this choice because it classifies both IL and SIL as under-approximation and they share the ability to drop program paths (e.g., by finite unrolling of loops).

5.3.1 Pairwise comparison

We first carry out a comparison of the logics two by two. We skip the pair HL and IL since it was already discussed when IL was introduced in [OHe20].

NC and IL

Sufficient preconditions are properties that imply Dijkstra's **wlp**: $\overline{P}$ is sufficient for a postcondition Q if and only if $\overline{P} \subseteq \mathbf{wlp}[\mathsf{r}](Q)$, which in turn is equivalent to validity of the HL triple $\{\overline{P}\}$ r $\{Q\}$. Necessary and sufficient preconditions are dual, and so are IL and HL. Moreover, NC and IL enjoy the same consequence rule: both can strengthen the post and weaken the pre. This double duality suggests a relation between NC and IL. However, the following example shows this is not the case.

Example 5.7. Consider the nondeterministic program $\mathsf{r42}$ from Example 2.8, where $Q_{42} \triangleq (z = 42)$. For brevity, we let $Q'_{42} \triangleq (Q_{42} \land \text{odd}(y) \land \text{even}(x))$. From that Example we know that $[z = 11]$ $\mathsf{r42}$ $[Q'_{42}]$ is valid in IL. However, we observe that the NC triple $(z = 11)$ $\mathsf{r42}$ (Q'_{42}) is not valid because, e.g., the state $[y \mapsto 1, z \mapsto 10]$ has an execution leading to Q'_{42} but doesn't satisfy $z = 11$. Moreover, take for instance $\underline{P} \triangleq \text{odd}(y)$, which makes the NC triple $(\underline{P})$ $\mathsf{r42}$ (Q'_{42}) valid (in any state *not* satisfying $\underline{P}$ y is even, is not changed by $\mathsf{r42}$ and should be odd to satisfy Q'_{42}). Then it is clear that $(z = 11) \not\Rightarrow \underline{P}$. This shows that not only IL triples do not yield NC triples, but also that in general there are NC preconditions which are not implied by IL preconditions.

Conversely, consider $\neg Q_{42} = (z \neq 42)$. While the NC triple $(\mathbf{true})$ $\mathsf{r42}$ $(\neg Q_{42})$ is valid, the IL triple $[\mathbf{true}]$ $\mathsf{r42}$ $[\neg Q_{42}]$ is not: for instance, the final state $[x \mapsto 11, y \mapsto 11, z \mapsto 11]$ is not reachable from any initial state. It follows that the IL triple $[P]$ $\mathsf{r42}$ $[\neg Q_{42}]$ is not valid for any P. ■

Given $\vDash (\underline{P})$ r (Q) and $\vDash [P]$ r $[Q]$, there always are states satisfying both P and $\underline{P}$, i.e., $P \cap \underline{P} \neq \emptyset$. However, in general neither $P \subseteq \underline{P}$ nor $\underline{P} \subseteq P$. The difference between NC and IL becomes apparent when we spell out their validity conditions using quantifiers:

$$\forall \sigma' \in Q \,.\, \forall \sigma \in [\![\overleftarrow{\mathsf{r}}]\!]\sigma' \,.\, \sigma \in P \qquad (\text{NC}_{\text{FOL}})$$

$$\forall \sigma' \in Q \,.\, \exists \sigma \in [\![\overleftarrow{\mathsf{r}}]\!]\sigma' \,.\, \sigma \in P \qquad (\text{IL}_{\text{FOL}})$$

Initial states are universally quantified in (NC_{FOL})—*all* initial states with a good run must satisfy the precondition—but they are existentially quantified in (IL_{FOL}). We also note that, when r is reversible (i.e., $[\![\mathsf{r}]\!]$ is injective) any valid IL triple is also a valid NC triple.

NC and HL

It turns out that NC is strongly connected to weakest liberal preconditions and thus to HL. Let Q be a correctness postcondition: a finite trace is in $\mathcal{T}(\sigma)$ if its final state satisfies Q and in $\mathcal{E}(\sigma)$ otherwise. In general, a necessary precondition has no relationship with $\mathbf{wlp}[\mathsf{r}](Q)$. However, if we consider $\neg Q$ instead of Q, we observe that "erroneous" executions becomes those in $\mathcal{T}(\sigma)$ and "correct" ones those in $\mathcal{E}(\sigma)$. This means that $\mathcal{T}(\sigma) = \emptyset$ iff $\sigma \in \mathbf{wlp}[\mathsf{r}](\neg Q)$, from which we derive $\neg\underline{P} \subseteq \mathbf{wlp}[\mathsf{r}](\neg Q)$ or, equivalently, $\neg\mathbf{wlp}[\mathsf{r}](\neg Q) \subseteq \underline{P}$.

Example 5.8. Consider again program $\mathsf{r42}$ from Example 2.8 and the correctness specification $\neg Q_{42} = (z \neq 42)$. We have that $\mathbf{wlp}[\mathsf{r42}](\neg\neg Q_{42}) = Q_{42}$, because if initially $z \neq 42$ then it is possible that x is assigned an odd value and z is not updated. Hence, a condition P is implied by $\neg\mathbf{wlp}[\mathsf{r}](\neg\neg Q_{42}) = \neg Q_{42}$ if and only if it is necessary. For instance, $(z \neq 42 \lor \text{odd}(y))$ is necessary but $(z > 42)$ is not. ■

The next bijection establishes the connection between (NC) and (HL): a necessary precondition is just the negation of a sufficient precondition for the negated post. This was also observed using weakest (liberal) preconditions in [ZK22, Theorem 5.4].

Proposition 5.9 (Bijection between NC and HL)**.** *For any* $\mathsf{r} \in \mathsf{Reg}$ *and* $P, Q \subseteq \Sigma$*:*

$$[\![\mathsf{r}]\!]P \subseteq Q \Leftrightarrow [\![\overleftarrow{\mathsf{r}}]\!](\neg Q) \subseteq \neg P.$$

SIL and IL

Proposition 5.9 highlights an isomorphism between (HL) and (NC). By duality, this naturally sparks the question whether a similar connections between (IL) and (SIL) exists. The next example shows this is not the case.

Example 5.10. Since IL and SIL enjoy different consequence rules, neither of the two can imply the other with the same P and Q. If we take negation into account, consider the program $\mathsf{r1} \triangleq$ `x := 1`. Both the SIL triple $\langle\!\langle x \geq 0 \rangle\!\rangle$ $\mathsf{r1}$ $\langle\!\langle x = 1 \rangle\!\rangle$ and the IL triple $[x \geq 0]$ $\mathsf{r1}$ $[x = 1]$ are valid. However, neither $[x < 0]$ $\mathsf{r1}$ $[x \neq 1]$ nor $\langle\!\langle x < 0 \rangle\!\rangle$ $\mathsf{r1}$ $\langle\!\langle x \neq 1 \rangle\!\rangle$ are valid. So neither (IL) implies negated (SIL) nor the other way around. ■

To gain some insights on why (SIL) and (IL) are not equivalent, we introduce the following concepts.

Definition 5.11. Given a regular command r, we define the set of states that only diverges D_r and the set of unreachable states U_r:

$$D_\mathsf{r} \triangleq \{\sigma \mid [\![\mathsf{r}]\!]\sigma = \emptyset\} \qquad U_\mathsf{r} \triangleq \{\sigma' \mid \sigma' \not\in [\![\mathsf{r}]\!]\Sigma\} = \{\sigma' \mid [\![\overleftarrow{\mathsf{r}}]\!]\sigma' = \emptyset\}.$$

These two definition are dual when we reverse the execution direction: if we consider $[\![\overleftarrow{\mathsf{r}}]\!]$ instead of $[\![\mathsf{r}]\!]$, the roles of D and U are swapped. This means that, in a sense, U_r is the set of states which "diverge" going backward.

Lemma 5.12. *For any regular command* $\mathsf{r} \in \mathsf{Reg}$ *and sets of states* $P, Q \subseteq \Sigma$ *it holds:*

1. $[\![\overleftarrow{\mathsf{r}}]\!][\![\mathsf{r}]\!]P \supseteq P \setminus D_\mathsf{r}$;
2. $[\![\mathsf{r}]\!][\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq Q \setminus U_\mathsf{r}$.

This lemma highlights the asymmetry between over and under-approximation: the composition of a function with its inverse is increasing (but for non-terminating states). This explains why (HL) and (NC) are related while (IL) and (SIL) are not: for over-approximation, $P \setminus D_\mathsf{r} \subseteq [\![\overleftarrow{\mathsf{r}}]\!][\![\mathsf{r}]\!]P$ can be further exploited if we know $[\![\mathsf{r}]\!]P \subseteq Q$ via (HL), but it cannot when $[\![\mathsf{r}]\!]P \supseteq Q$ via (IL).

Lastly, while IL and SIL are not directly comparable, the preprint [RVO24] introduces a forward-oriented proof system with a core set of rules that are sound for both IL and SIL, therefore proving only triples that are valid for both. It also becomes complete for IL, resp. SIL, when augmented with the corresponding consequence rule.

SIL and HL

In general, HL and SIL are different, but they coincide whenever the program r is deterministic and terminates for every input.

Proposition 5.13. *For any* $\mathsf{r} \in \mathsf{Reg}$ *and* $P, Q \subseteq \Sigma$*:*

- *if* r *is deterministic,* $[\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P \Rightarrow [\![\mathsf{r}]\!]P \subseteq Q$
- *if* r *is terminating,* $[\![\mathsf{r}]\!]P \subseteq Q \Rightarrow [\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P$

Example 5.14. From Example 2.8, we know $\{\mathrm{odd}(y)\}$ $\mathsf{r42}$ $\{Q_{42}\}$ is a valid HL triple. Moreover, $\mathsf{r42}$ always terminates, so according to Proposition 5.13 $\langle\!\langle \mathrm{odd}(y) \rangle\!\rangle$ $\mathsf{r42}$ $\langle\!\langle Q_{42} \rangle\!\rangle$ is valid. Indeed, whenever y is odd in the initial state, x can be assigned nondeterministically an even value so that execution enters the if statement and z is assigned 42. ■

Rule	SIL	HL	IL
atom	$\frac{}{\langle\!\langle [\![\overleftarrow{\mathsf{c}}]\!]Q \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle Q \rangle\!\rangle}$	$\frac{}{\{P\}\ \mathsf{c}\ \{[\![\mathsf{c}]\!]P\}}$	$\frac{}{[P]\ \mathsf{c}\ [[\![\mathsf{c}]\!]P]}$
cons	$\frac{P \subseteq P' \quad \langle\!\langle P' \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q' \rangle\!\rangle \quad Q' \subseteq Q}{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}$	$\frac{P \subseteq P' \quad \{P'\}\ \mathsf{r}\ \{Q'\} \quad Q' \subseteq Q}{\{P\}\ \mathsf{r}\ \{Q\}}$	$\frac{P \supseteq P' \quad [P']\ \mathsf{r}\ [Q'] \quad Q' \supseteq Q}{[P]\ \mathsf{r}\ [Q]}$
seq	$\frac{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle R \rangle\!\rangle \quad \langle\!\langle R \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}_1;\mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}$	$\frac{\{P\}\ \mathsf{r}_1\ \{R\} \quad \{R\}\ \mathsf{r}_2\ \{Q\}}{\{P\}\ \mathsf{r}_1;\mathsf{r}_2\ \{Q\}}$	$\frac{[P]\ \mathsf{r}_1\ [R] \quad [R]\ \mathsf{r}_2\ [Q]}{[P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}$
choice	$\frac{\forall i \in \{1,2\} \quad \langle\!\langle P_i \rangle\!\rangle\ \mathsf{r}_i\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P_1 \cup P_2 \rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}$	$\frac{\forall i \in \{1,2\} \quad \{P\}\ \mathsf{r}_i\ \{Q\}}{\{P\}\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \{Q\}}$	$\frac{\forall i \in \{1,2\} \quad [P]\ \mathsf{r}_i\ [Q_i]}{[P]\ \mathsf{r}_1 \oplus \mathsf{r}_2\ [Q_1 \cup Q_2]}$
iter	$\frac{\forall n \geq 0 \,.\, \langle\!\langle Q_{n+1} \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_n \rangle\!\rangle}{\langle\!\langle \bigcup_{n \geq 0} Q_n \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q_0 \rangle\!\rangle}$	$\frac{\{P\}\ \mathsf{r}\ \{P\}}{\{P\}\ \mathsf{r}^\star\ \{P\}}$	$\frac{\forall n \geq 0 \,.\, [P_n]\ \mathsf{r}\ [P_{n+1}]}{[P_0]\ \mathsf{r}^\star\ [\bigcup_{n \geq 0} P_n]}$
empty	$\frac{}{\langle\!\langle \emptyset \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}$	$\frac{}{\{\emptyset\}\ \mathsf{r}\ \{Q\}}$	$\frac{}{[P]\ \mathsf{r}\ [\emptyset]}$
disj	$\frac{\langle\!\langle P_1 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_1 \rangle\!\rangle \quad \langle\!\langle P_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_2 \rangle\!\rangle}{\langle\!\langle P_1 \cup P_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_1 \cup Q_2 \rangle\!\rangle}$	$\frac{\{P_1\}\ \mathsf{r}\ \{Q_1\} \quad \{P_2\}\ \mathsf{r}\ \{Q_2\}}{\{P_1 \cup P_2\}\ \mathsf{r}\ \{Q_1 \cup Q_2\}}$	$\frac{[P_1]\ \mathsf{r}\ [Q_1] \quad [P_2]\ \mathsf{r}\ [Q_2]}{[P_1 \cup P_2]\ \mathsf{r}\ [Q_1 \cup Q_2]}$
iter0	$\frac{}{\langle\!\langle Q \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q \rangle\!\rangle}$	unsound	$\frac{}{[P]\ \mathsf{r}^\star\ [P]}$
unroll	$\frac{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}^\star;\mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}{\langle\!\langle P \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q \rangle\!\rangle}$	unsound	$\frac{[P]\ \mathsf{r}^\star;\mathsf{r}\ [Q]}{[P]\ \mathsf{r}^\star\ [Q]}$
conj	unsound	$\frac{\{P_1\}\ \mathsf{r}\ \{Q_1\} \quad \{P_2\}\ \mathsf{r}\ \{Q_2\}}{\{P_1 \cap P_2\}\ \mathsf{r}\ \{Q_1 \cap Q_2\}}$	unsound

Figure 5.5: Comparison of SIL, HL and IL rules. Identical rules are highlighted in purple.

5.3.2 Inference rules

In Figure 5.5 we compare the rules of SIL, HL and IL, so to emphasize the similarities and differences among them. HL rule {iter} says that any invariant is acceptable, not necessarily the minimal one, so that HL relies on over-approximation. This is confirmed by the rows for cons and empty, where on the contrary IL and SIL are shown to rely on under-approximation. The consequence rule is the key rule of all the logics, because it allows to generalize a proof by weakening/strengthening the two conditions P and Q involved. The direction of rules ⟪cons⟫ of SIL and {cons} of HL is the same and it is exactly the opposite direction of rule [cons] of IL and NC, which coincide. So the different consequence rules follow the diagonals of Figure 5.1. The row for rules seq and disj show that in all cases triples can be composed sequentially and additively. Rules iter0 and unroll correspond to finite loop unrolling and are a prerogative of under-approximation: they are the same for SIL and IL, but they are unsound for HL.

The atom rule deserves a more in-depth discussion. The presented version using sets shows that HL and IL exploit the forward semantics, and SIL the backward one. However, if we instead use formulae as pre and postconditions, this rule must be instantiated for all atomic construct, particularly for assignments. It is well known that there are two different, valid axioms for assignment in HL: Hoare's backward substitution [Hoa69] and Floyd's forward inference [Flo67] (where $q[a/x]$ denotes the usual capture-avoiding substitution of all free occurrences of x in q with the expression a).

$$\frac{}{\{q[a/x]\}\ \texttt{x := a}\ \{q\}}\ \{\mathsf{Hoare}\} \qquad \frac{}{\{p\}\ \texttt{x := a}\ \{\exists x'.p[x'/x] \wedge x = a[x'/x]\}}\ \{\mathsf{Floyd}\}$$

While both axioms are valid in HL, only Floyd's forward axiom is valid in IL [OHe20, §4], already showing a broken symmetry between over and under-approximation. While our presentation of SIL rules uses sets of states, we use Hoare's backward substitution in Separation SIL (see Figure 5.7): dually to the forward IL, the backward axiom is valid for SIL. Surprisingly, Floyd's forward axiom is valid in SIL as well: this shows that even for under-approximation, forward and backward semantics behave differently. This is possibly rooted in the properties of arithmetic expressions: they are defined for every input but not necessarily surjective. In the terminology of Lemma 5.12, $D_{\mathtt{x:=a}}$ is always empty but $U_{\mathtt{x:=a}}$ may not be.

5.3.3 Weakest/strongest conditions

Depending on the way in which program analysis is conducted, it can be interesting to derive either the most general or most specific hypotheses for the given property. For instance, given a correctness specification Q, one is typically interested in finding the minimal constraint on the input that guarantees program correctness (this correspond to computing Dijkstra's **wlp**). Conversely, to infer necessary conditions we can be interested in devising the strongest hypotheses under which some correct run is possible.

To investigate the existence of weakest/strongest pre and post, we find convenient to focus on the validity of the four kinds of triples as shown in Figure 5.1. The concrete semantics is trivially a strongest (HL and NC) or weakest (IL and SIL) condition for the "target" property (i.e., P computing backward and Q forward). However, it turns out that having a best condition on the "source" property is a prerogative of over-approximation, i.e., that over and under-approximation are not dual theories in this respect.

Proposition 5.15 (Existence of weakest conditions). *For any command* $\mathsf{r} \in \mathsf{Reg}$*:*

- *given Q, there exists a weakest P such that $[\![\mathsf{r}]\!]P \subseteq Q$ (HL);*
- *given P, there exists a weakest Q such that $[\![\overleftarrow{\mathsf{r}}]\!]Q \subseteq P$ (NC).*

Proposition 5.16 (Non-existence of strongest conditions). *For any command* $\mathsf{r} \in \mathsf{Reg}$*:*

- *for some Q, there is no strongest P such that $[\![\mathsf{r}]\!]P \supseteq Q$ (IL);*
- *for some P, there is no strongest Q such that $[\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq P$ (SIL).*

The reason why strongest conditions may not exist for IL and SIL is that collecting semantics (both forward and backward) are additive but not co-additive. In other words, rule $\{\mathsf{disj}\}$ is sound for all triples, while rule $\{\mathsf{conj}\}$ is valid for HL and NC but neither for IL nor SIL, as shown in Figure 5.5. This means that given two IL triples $[P_1]\ \mathsf{r}\ [Q]$ and $[P_2]\ \mathsf{r}\ [Q]$, in general $[\![\mathsf{r}]\!](P_1 \cap P_2) \not\supseteq Q$ in which case $[P_1 \cap P_2]\ \mathsf{r}\ [Q]$ is not valid, as shown in the following example.

Example 5.17. Consider the program $\mathsf{r1} \triangleq x := 1$. The two IL triples $[x = 0]\ \mathsf{r1}\ [x = 1]$ and $[x = 10]\ \mathsf{r1}\ [x = 1]$ are valid, but their intersection is $[\emptyset]\ \mathsf{r1}\ [x = 1]$, which is not valid.

For SIL, consider the program $\mathsf{rnd} \triangleq$ `x := nondet()`. Both triples $\langle\!\langle x = 1 \rangle\!\rangle\ \mathsf{rnd}\ \langle\!\langle x = 0 \rangle\!\rangle$ and $\langle\!\langle x = 1 \rangle\!\rangle\ \mathsf{rnd}\ \langle\!\langle x = 10 \rangle\!\rangle$ are valid, but also incomparable and minimal because $\emptyset$ is not a valid postcondition. ■

This can also be observed using the theory of adjunction. It is well known that left adjoints are additive while right adjoints are co-additive [DP02]. The weakest precondition **wlp** for HL is characterized by the adjunctive property $P \subseteq \mathbf{wlp}[\mathsf{r}](Q)$ iff $[\![\mathsf{r}]\!]P \subseteq Q$ (weakest postconditions for NC are defined analogously). Since the forward (resp. backward) semantics is additive we get the existence of its right adjoint, that is exactly HL weakest precondition (resp. NC weakest postcondition). However, a strongest precondition **sp** for IL would satisfy the adjunctive property $\mathbf{sp}[\mathsf{r}](Q) \subseteq P$ iff $Q \subseteq [\![\mathsf{r}]\!]P$, making the non co-additive forward semantics a right adjoint. Similarly, a strongest postcondition for SIL would be a left adjoint of the backward semantics.

5.3.4 Termination and Reachability

Termination and reachability are two sides of the same coin when switching from forward to backward semantics, and over and under-approximation behave differently with respect to these notions.

For HL we can only distinguish a precondition which always causes divergence: if $\{P\}$ r $\{\emptyset\}$, all states in the precondition P will always diverge. However, if just one state in P has one terminating computation, its final state must be in $Q \neq \emptyset$, so we cannot say whether states in P diverge or not. Moreover, because of the over-approximation, a non empty Q does not mean there truly are finite executions, as those may be introduced by the approximation. Dually, NC cannot say much about reachability of Q unless P is empty, in which case Q is unreachable.

On the contrary, under-approximation offers much stronger guarantees on termination/reachability. Any IL triple $[P]$ r $[Q]$ ensures that all states in Q are reachable (in particular, from states in P). Dually, a SIL triple $\langle\!\langle P \rangle\!\rangle$ r $\langle\!\langle Q \rangle\!\rangle$ means that all states in P have a convergent computation that ends in some state in Q. This observation motivates the design of a forward iteration rule in IL (resp. backward in SIL): a backward (resp. forward) rule would need to prove reachability of all points in the post (resp. pre). Instead, the forward rule of IL (resp. backward rule of SIL) ensures reachability (resp. termination) by construction, as it builds Q (resp. P) only with points known to be reachable (resp. terminating).

5.4 Separation Sufficient Incorrectness Logic

We instantiate SIL to handle pointers and dynamic memory allocation, introducing Separation SIL. The goal of Separation SIL is to identify the causes of memory errors: it takes the backward under-approximation principles of SIL and combines it with the ability to deal with pointers from Separation Logic (SL) [Rey02; ORY01]

5.4.1 Heap regular commands

We denote by HRCmd the set of heap regular commands obtained by plugging the following definition of heap atomic commands in (2.3) (in blue new primitives):

$$\mathsf{HACmd} \ni \mathsf{c} ::= \texttt{skip} \mid \texttt{x := a} \mid \texttt{b?} \mid \texttt{x := alloc()} \mid \texttt{free(x)} \mid \texttt{x := [y]} \mid \texttt{[x] := y}$$

where we assume that `x` and `y` are syntactically distinct variables. The command `x := alloc()` allocates a new memory location containing a nondeterministic value, `free(x)` deallocates memory, and `[·]` dereferences a variable. The syntax only allows to allocate, free and dereference single variables. To use a value from the heap in an arithmetic $\mathsf{a} \in \mathsf{AExp}$ or Boolean expressions $\mathsf{b} \in \mathsf{BExp}$, it must be loaded in a variable beforehand.

Given a heap command $\mathsf{r} \in \mathsf{HRCmd}$, we let $\mathrm{fv}(\mathsf{r}) \subseteq \mathrm{Var}$ be the set of (free) variables of r and $\mathrm{mod}(\mathsf{r}) \subseteq \mathrm{Var}$ be the set of variables modified by `r`. The definition of the former is standard, while the latter is defined inductively in Figure 5.6. Note that `free(x)` and `[x] := y` do not modify `x`: they only modify the value *pointed by* `x`, not the actual value of `x` (the memory address itself).

$$\begin{array}{ll}
\mathrm{mod}(\texttt{skip}) = \emptyset & \mathrm{mod}(\texttt{x := a}) = \{\texttt{x}\} \\
\mathrm{mod}(\texttt{b?}) = \emptyset & \mathrm{mod}(\texttt{x := alloc()}) = \{\texttt{x}\} \\
\mathrm{mod}(\texttt{free(x)}) = \emptyset & \mathrm{mod}(\texttt{x := [y]}) = \{\texttt{x}\} \\
\mathrm{mod}(\texttt{[x] := y}) = \emptyset & \mathrm{mod}(\mathsf{r}_1; \mathsf{r}_2) = \mathrm{mod}(\mathsf{r}_1) \cup \mathrm{mod}(\mathsf{r}_2) \\
\mathrm{mod}(\mathsf{r}_1 \oplus \mathsf{r}_2) = \mathrm{mod}(\mathsf{r}_1) \cup \mathrm{mod}(\mathsf{r}_2) & \mathrm{mod}(\mathsf{r}^\star) = \mathrm{mod}(\mathsf{r})
\end{array}$$

Figure 5.6: Definition of $\mathrm{mod}(\mathsf{r})$.

5.4.2 Assertion language

Our assertion language is derived from SL (see Section 2.7) and ISL [Raa+20]:

$$\mathrm{Asl} \ni p, q ::= \mathbf{false} \mid \mathbf{true} \mid p \wedge q \mid p \vee q \mid \exists x.p \mid \mathsf{a} \asymp \mathsf{a} \mid \mathbf{emp} \mid x \mapsto \mathsf{a} \mid x \not\mapsto \ \mid p * q$$

where $\asymp \in \{=, \neq, \leq, <, \dots\}$ replaces standard comparison operators, $x \in \mathrm{Var}$ is a generic variable and $\mathsf{a} \in \mathsf{AExp}$ is an arithmetic expression. The first six constructs describe a fragment of first-order logic, called coherent logic [BC05], which is also the one used in bi-abduction [CDOY09]. The last four describe heaps. **emp** denotes an empty heap, $x \mapsto a$ represents an heap with a single memory cell pointed by x and whose content is a, $x \not\mapsto$ describes that x points to a previously deallocated memory cell (it was first introduced in [Raa+20]). The separating conjunction $p * q$ is a key feature of Separation Logics and describes an heap which can be divided in two disjoint sub-heaps, one satisfying p and the other q. We let $x \mapsto - \triangleq \exists v.x \mapsto v$ describe that x is allocated without tracking its exact value. Given a formula $p \in \mathrm{Asl}$, we call $\mathrm{fv}(p) \subseteq \mathrm{Var}$ the set of its free variables.

5.4.3 Proof system

We present the rules of Separation SIL in Figure 5.7. $q[a/x]$ is the capture-avoiding substitution. For the sake of presentation, we present rules without explicit error management (see Remark 2.9). However, the extension is straightforward: in Section 5.4.6 we present the error rule for store and its use in Example 5.22, as well as discussing the formal changes to the semantics model.

We split the rules in three groups. The first group gives the rules for atomic commands $\mathsf{c} \in \mathsf{HACmd}$, i.e., all instances of the SIL rule ⟪atom⟫. The second one includes rules borrowed from SL, the third one from SIL.

Rule ⟪skip⟫ is straightforward: whatever is true before and after the `skip` can be added with ⟪frame⟫. Rule ⟪assign⟫ is Hoare's backward assignment rule [Hoa69]. Floyd's forward axiom [Flo67] is also valid for SIL (see Section 5.3.2), but we opt for Hoare's rule because it fits better with the backward analysis of SIL. Rule ⟪assume⟫ conjoins the assertion b to the postcondition: only states satisfying the Boolean guard can reach the post. Rule ⟪alloc⟫ allocates a new memory location for x. The premise is empty: if the previous content of x is needed, $x = z$ can be introduced in the premise with ⟪cons⟫. Rule ⟪free⟫ requires x to be allocated before freeing it. Rule ⟪load⟫ is similar to ⟪assign⟫, with the addition of the (disjoint) $y \mapsto a$ to make sure that y is allocated. Rule ⟪store⟫ requires that x is allocated, and updates the value it points to. All these rules are local: thanks to ⟪frame⟫, they can specify only pre and post for the modified part of the heap.

Rule ⟪exists⟫ allows to "hide" local variables. Rule ⟪frame⟫ is typical of separation logics [Rey02; Raa+20]: it allows to add a frame around a derivation through the separating

$$\frac{}{\langle\!\langle \mathbf{emp} \rangle\!\rangle\ \texttt{skip}\ \langle\!\langle \mathbf{emp} \rangle\!\rangle}\ \langle\!\langle \mathsf{skip} \rangle\!\rangle \qquad \frac{}{\langle\!\langle q[a/x] \rangle\!\rangle\ \texttt{x := a}\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{assign} \rangle\!\rangle$$

$$\frac{}{\langle\!\langle q \land b \rangle\!\rangle\ \texttt{b?}\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{assume} \rangle\!\rangle \qquad \frac{}{\langle\!\langle \mathbf{emp} \rangle\!\rangle\ \texttt{x := alloc()}\ \langle\!\langle x \mapsto v \rangle\!\rangle}\ \langle\!\langle \mathsf{alloc} \rangle\!\rangle$$

$$\frac{}{\langle\!\langle x \mapsto - \rangle\!\rangle\ \texttt{free(x)}\ \langle\!\langle x \not\mapsto \rangle\!\rangle}\ \langle\!\langle \mathsf{free} \rangle\!\rangle \qquad \frac{x \notin \mathrm{fv}(a)}{\langle\!\langle y \mapsto a * q[a/x] \rangle\!\rangle\ \texttt{x := [y]}\ \langle\!\langle y \mapsto a * q \rangle\!\rangle}\ \langle\!\langle \mathsf{load} \rangle\!\rangle$$

$$\frac{}{\langle\!\langle x \mapsto - \rangle\!\rangle\ \texttt{[x] := y}\ \langle\!\langle x \mapsto y \rangle\!\rangle}\ \langle\!\langle \mathsf{store} \rangle\!\rangle$$

$$\frac{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle \quad x \notin \mathrm{fv}(\mathsf{r})}{\langle\!\langle \exists x.p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle \exists x.q \rangle\!\rangle}\ \langle\!\langle \mathsf{exists} \rangle\!\rangle \qquad \frac{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle \quad \mathrm{fv}(t) \cap \mathrm{mod}(\mathsf{r}) = \emptyset}{\langle\!\langle p * t \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q * t \rangle\!\rangle}\ \langle\!\langle \mathsf{frame} \rangle\!\rangle$$

$$\frac{p \Rightarrow p' \quad \langle\!\langle p' \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q' \rangle\!\rangle \quad q' \Rightarrow q}{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{cons} \rangle\!\rangle \qquad \frac{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle t \rangle\!\rangle \quad \langle\!\langle t \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q \rangle\!\rangle}{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}_1; \mathsf{r}_2\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{seq} \rangle\!\rangle$$

$$\frac{\langle\!\langle p_1 \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle q \rangle\!\rangle \quad \langle\!\langle p_2 \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q \rangle\!\rangle}{\langle\!\langle p_1 \lor p_2 \rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{choice} \rangle\!\rangle \qquad \frac{\forall n \geq 0\ \ \langle\!\langle q(n+1) \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q(n) \rangle\!\rangle}{\langle\!\langle \exists n.q(n) \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q(0) \rangle\!\rangle}\ \langle\!\langle \mathsf{iter} \rangle\!\rangle$$

$$\frac{}{\langle\!\langle \mathbf{false} \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{empty} \rangle\!\rangle \qquad \frac{\langle\!\langle p_1 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q_1 \rangle\!\rangle \quad \langle\!\langle p_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q_2 \rangle\!\rangle}{\langle\!\langle p_1 \lor p_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q_1 \lor q_2 \rangle\!\rangle}\ \langle\!\langle \mathsf{disj} \rangle\!\rangle$$

$$\frac{}{\langle\!\langle q \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{iter0} \rangle\!\rangle \qquad \frac{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}^\star; \mathsf{r}\ \langle\!\langle q \rangle\!\rangle}{\langle\!\langle p \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{unroll} \rangle\!\rangle$$

Figure 5.7: Proof rules for Separation SIL. The first group replaces SIL rule ⟪atom⟫, the second includes rules peculiar of SL, the third includes rule from SIL.

conjunction $*$, plugging the proof for a small portion of a program inside a larger heap. In the third group, we instantiated the SIL rules from Figure 5.3 for logical formulae, by replacing set theoretical symbols (such as $\subseteq$ and $\emptyset$) with the corresponding logical symbols (such as $\Rightarrow$ and **false**, respectively). The only notable difference is in rule ⟪iter⟫, where Separation SIL uses a predicate $q(n)$ parametrized by the natural number $n \in \mathbb{N}$ and the precondition $\exists n.q(n)$ in the conclusion of the rule. This is a logical replacement for the infinite union used in SIL rule.

5.4.4 Soundness

To prove soundness of Separation SIL, we give a semantic model for heap regular commands. Fixed a finite set Var of variables and an infinite set Loc of memory locations, we define the set of values as Val $\triangleq \mathbb{Z} \uplus$ Loc ($\uplus$ is disjoint union). Stores $s \in$ Store are (total) functions $s : \mathrm{Var} \to \mathrm{Val}$; heaps $h \in$ Heap are partial functions $h : \mathrm{Loc} \rightharpoonup \mathrm{Val} \uplus \{\delta\}$. If $h(l) = v \in$ Val, location l is allocated and holds value v, if $l \notin \mathrm{dom}(h)$ then it is not allocated. The special value δ describes a deallocated memory location: if $h(l) = \delta$, that location was previously allocated and then deallocated. As notation, we use $s[x \mapsto v]$ for function update, $[]$ for the empty heap and $[l \mapsto v]$ for the heap defined only on l and associating value v to it. We say two heaps are disjoint, written $h_1 \perp h_2$, when $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$, and in that case we define the $\bullet$ operation as the merge of

$$
\begin{aligned}
\llparenthesis \texttt{skip} \rrparenthesis (s,h) &\triangleq \{(s,h)\} \\
\llparenthesis \texttt{x := a} \rrparenthesis (s,h) &\triangleq \{(s[x \mapsto \llparenthesis \mathsf{a} \rrparenthesis s], h)\} \\
\llparenthesis \texttt{b?} \rrparenthesis (s,h) &\triangleq \begin{cases} \{(s,h)\} & \text{if } \llparenthesis \mathsf{b} \rrparenthesis s = \texttt{tt} \\ \emptyset & \text{otherwise} \end{cases} \\
\llparenthesis \texttt{x := alloc()} \rrparenthesis (s,h) &\triangleq \{(s[x \mapsto l], h[l \mapsto v]) \mid v \in \mathrm{Val}, \mathit{avail}(l)\} \\
\llparenthesis \texttt{free(x)} \rrparenthesis (s,h) &\triangleq \{(s, h[s(x) \mapsto \delta])\} \quad \text{if } h(s(x)) \in \mathrm{Val} \\
\llparenthesis \texttt{x := [y]} \rrparenthesis (s,h) &\triangleq \{(s[x \mapsto h(s(y))], h)\} \quad \text{if } h(s(y)) \in \mathrm{Val} \\
\llparenthesis \texttt{[x] := y} \rrparenthesis (s,h) &\triangleq \{(s, h[s(x) \mapsto s(y)])\} \quad \text{if } h(s(x)) \in \mathrm{Val}
\end{aligned}
$$

(a) Semantics of heap atomic commands, where $\mathit{avail}(l) \triangleq (l \notin \mathrm{dom}(h) \vee h(l) = \delta)$ and we assume that $\llparenthesis \mathsf{c} \rrparenthesis (s,h) \triangleq \{\mathbf{err}\}$ unless differently stated.

$$
\begin{aligned}
\{\!| a_1 \asymp a_2 |\!\} &\triangleq \{(s,h) \mid \llparenthesis a_1 \rrparenthesis s \asymp \llparenthesis a_2 \rrparenthesis s\} & \{\!| \mathbf{false} |\!\} &\triangleq \emptyset \\
\{\!| \exists x.p |\!\} &\triangleq \{(s,h) \mid \exists v \in \mathrm{Val}\,.\,(s[x \mapsto v], h) \in \{\!| p |\!\}\} & \{\!| \mathbf{true} |\!\} &\triangleq \Sigma \\
\{\!| x \not\mapsto\ |\!\} &\triangleq \{(s, [s(x) \mapsto \delta])\} & \{\!| p \vee q |\!\} &\triangleq \{\!| p |\!\} \cup \{\!| q |\!\} \\
\{\!| x \mapsto a |\!\} &\triangleq \{(s, [s(x) \mapsto \llparenthesis a \rrparenthesis s])\} & \{\!| p \wedge q |\!\} &\triangleq \{\!| p |\!\} \cap \{\!| q |\!\} \\
\{\!| p * q |\!\} &\triangleq \{(s, h_p \bullet h_q) \mid (s,h_p) \in \{\!| p |\!\}, (s,h_q) \in \{\!| q |\!\}, h_p \perp h_q\} & \{\!| \mathbf{emp} |\!\} &\triangleq \{(s, [])\}
\end{aligned}
$$

(b) Semantics of the assertion language.

Figure 5.8: Ingredients to prove soundness of Separation SIL.

the two: $h_1 \bullet h_2$ coincides with h_1 on $\mathrm{dom}(h_1)$, with h_2 on $\mathrm{dom}(h_2)$ and it is undefined everywhere else.

Let $\Sigma = \mathrm{Store} \times \mathrm{Heap}$, and $\Sigma_e = \Sigma \uplus \{\mathbf{err}\}$: states $\sigma \in \Sigma_e$ are either a pair store/heap or the error state $\mathbf{err}$. The denotational semantics of atomic commands $\llparenthesis \cdot \rrparenthesis : \mathsf{HACmd} \to \wp(\Sigma_e) \to \wp(\Sigma_e)$ is in Figure 5.8a. To simplify the presentation, we define it as $\llparenthesis \cdot \rrparenthesis : \mathsf{HACmd} \to \Sigma \to \wp(\Sigma_e)$, we let $\llparenthesis \mathsf{c} \rrparenthesis \mathbf{err} = \{\mathbf{err}\}$, and we lift it to set of states by union. Please note that evaluation of arithmetic a and Boolean expressions b only depends on the store since they cannot dereference variables. We define the forward collecting semantics of heap commands $\llbracket \cdot \rrbracket : \mathsf{HRCmd} \to \wp(\Sigma_e) \to \wp(\Sigma_e)$ just as in Figure 2.1 using the different semantics of atomic commands for $\mathsf{c} \in \mathsf{HACmd}$.

The semantics $\{\!| \cdot |\!\}$ of a formula $p \in \mathrm{Asl}$ is a set of states in Σ, and is defined in Figure 5.8b.

We say a Separation SIL triple $\langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle$ is valid if $\llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!| q |\!\} \supseteq \{\!| p |\!\}$. To prove soundness of Separation SIL, we rely on a stronger lemma, whose proof is by induction on the derivation tree. Then, by taking $t = \mathbf{emp}$ and using $p * \mathbf{emp} \equiv p$, we get the soundness of the proof system.

Lemma 5.18. *Let $p, q, t \in \mathit{Asl}$ and $\mathsf{r} \in \mathsf{HRCmd}$. If $\vdash \langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle$ and $\mathit{fv}(t) \cap \mathit{mod}(\mathsf{r}) = \emptyset$,*

$$\llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!| q * t |\!\} \supseteq \{\!| p * t |\!\}$$

Corollary 5.19 (Separation SIL is sound)**.** *Any provable Separation SIL triple is valid:*

$$\vdash \langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle \implies \vDash \langle\!\langle p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q \rangle\!\rangle$$

5.4.5 Example of Separation SIL derivation

We show in the next example how Separation SIL proof system can infer preconditions ensuring that a provided error can happen.

Example 5.20. Consider the the motivating example of [Raa+20], encoding a use-after-free bug involving C++ vector push_back function:

```
// program rclient                push_back(v) {
x := *v;                              if (nondet()) {
push_back(v);                             free(*v);
*x := 1;                                  *v := alloc();
                                  }   }
```

We encode the above program as a regular command by letting:

$$\mathsf{rclient} \triangleq x := [v];\ (\mathsf{r}_b \oplus \mathtt{skip}) \qquad\qquad \mathsf{r}_b \triangleq y := [v];\ \mathrm{free}(y);\ y := \mathrm{alloc}();\ [v] := y$$

Since our syntax does not include functions, we inline `push_back`. We cannot free and allocate $*v$ directly, whence the auxiliary variable y. For simplicity, we do not include the last assignment `*x := 1` in rclient: whenever the postcondition $x \not\mapsto$ holds, an error occurs after rclient.

We prove the Separation SIL triple

$$\langle\!\langle v \mapsto z * z \mapsto - * \mathbf{true} \rangle\!\rangle \ \mathsf{rclient}\ \langle\!\langle x \not\mapsto\ * \mathbf{true} \rangle\!\rangle$$

which ensures that *every* state in the precondition reaches the error, thus giving (many) actual witnesses for testing and debugging purposes. Moreover, Separation SIL proof system guides the crafting of the precondition if the proof is done from the error postcondition backward.

Let us fix the following assertions:

$$p \triangleq (v \mapsto z * z \mapsto - * \mathbf{true}) \quad q \triangleq (x \not\mapsto * \mathbf{true}) \quad t \triangleq (v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto) * \mathbf{true})$$

To prove the Separation SIL triple $\langle\!\langle p \rangle\!\rangle$ rclient $\langle\!\langle q \rangle\!\rangle$, we first prove the triple $\langle\!\langle t \rangle\!\rangle$ r_b $\langle\!\langle q \rangle\!\rangle$, whose derivation is in Figure 5.9b. Derivations are best read bottom-up: we start from the post and, for every atomic command, we find a suitable pre to apply the rule. In all cases, we strengthen the post to be able to apply the right rule: this usually means adding some constraint on the shape of the heap. Particularly, to apply the rule ⟪free⟫ we need y to be deallocated, and this can happen in two different ways: either if $y = x$, since x is deallocated; or if y is a new name. This is captured by the disjunction $x = y \vee x \not\mapsto$. We remark that this can be inferred algorithmically via Lemma 5.24.

Using the derivation in Figure 5.9b, we complete the proof as shown in Figure 5.9a. In the derivation, using rule ⟪load⟫ for the first assignment `x := [v]`, we get the pre $(v \mapsto z * z \mapsto - * (z = z \vee z \not\mapsto) * \mathbf{true})$, but since $z \mapsto - * z \not\mapsto$ is not satisfiable we remove that disjunct and find p.

Note the use of rule ⟪cons⟫ in the pre of the nondeterministic choice to remove the disjunct $(x \not\mapsto\ * \mathbf{true})$, effectively dropping the analysis of that program path. This correspond to IL's ability to drop paths going forward. ■

In the example, we use as error postcondition $x \not\mapsto\ * \mathbf{true}$. It is necessary to include $(* \mathbf{true})$ because, in final reachable states, x is not the only variable allocated (there are also v and y), so the final heap should talk about them as well. Adding $(* \mathbf{true})$ is a

$$
\begin{array}{l}
\langle\!\langle p : v \mapsto z * z \mapsto - * \mathbf{true} \rangle\!\rangle \\
\quad\ \ \equiv \\
\langle\!\langle v \mapsto z * (z = z \vee z \not\mapsto\ ) * z \mapsto - * \mathbf{true} \rangle\!\rangle \\
\quad x := [v] \\
\langle\!\langle \underline{v \mapsto z * (x = z \vee x \not\mapsto\ )} * z \mapsto - * \mathbf{true} \rangle\!\rangle \\
\langle\!\langle (\mathbf{true} * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto\ )) \vee q \rangle\!\rangle \\
\left(\begin{array}{l}
\langle\!\langle t \rangle\!\rangle \\
\quad y := [v]; \\
\quad \mathrm{free}(y); \\
\quad y := \mathrm{alloc}(); \\
\quad [v] := y \\
\langle\!\langle q \rangle\!\rangle
\end{array} \right)
\oplus
\begin{array}{l}
\langle\!\langle x \not\mapsto\ * \mathbf{true} \rangle\!\rangle \\
\quad \texttt{skip} \\
\langle\!\langle x \not\mapsto\ * \mathbf{true} \rangle\!\rangle
\end{array} \\
\langle\!\langle q : x \not\mapsto\ * \mathbf{true} \rangle\!\rangle
\end{array}
$$

(a) Linearized derivation of the Separation SIL triple $\langle\!\langle p \rangle\!\rangle$ $\mathsf{rclient}$ $\langle\!\langle q \rangle\!\rangle$. The omitted sub-derivation is in Figure 5.9b.

$$
\begin{array}{l}
\langle\!\langle t : \mathbf{true} * v \mapsto z * z \mapsto - * (x = z \vee x \not\mapsto\ ) \rangle\!\rangle \\
\quad y := [v]; \\
\langle\!\langle \mathbf{true} * \underline{v \mapsto z * y \mapsto - * (x = y \vee x \not\mapsto\ )} \rangle\!\rangle \\
\langle\!\langle \mathbf{true} * v \mapsto - * y \mapsto - * (x = y \vee x \not\mapsto\ ) \rangle\!\rangle \\
\quad \mathrm{free}(y); \\
\langle\!\langle \mathbf{true} * v \mapsto - * \underline{y \not\mapsto\ } * (x = y \vee x \not\mapsto\ ) \rangle\!\rangle \\
\langle\!\langle x \not\mapsto\ * v \mapsto - * \mathbf{emp} * \mathbf{true} \rangle\!\rangle \\
\quad y := \mathrm{alloc}(); \\
\langle\!\langle x \not\mapsto\ * v \mapsto - * \underline{y \mapsto y'} * \mathbf{true} \rangle\!\rangle \\
\langle\!\langle x \not\mapsto\ * v \mapsto - * \mathbf{true} \rangle\!\rangle \\
\quad [v] := y \\
\langle\!\langle x \not\mapsto\ * \underline{v \mapsto y} * \mathbf{true} \rangle\!\rangle \\
\langle\!\langle q : x \not\mapsto\ * \mathbf{true} \rangle\!\rangle
\end{array}
$$

(b) Linearized derivation of the Separation SIL triple $\langle\!\langle t \rangle\!\rangle$ r_b $\langle\!\langle q \rangle\!\rangle$.

Figure 5.9: The full derivation of $\langle\!\langle p \rangle\!\rangle$ $\mathsf{rclient}$ $\langle\!\langle q \rangle\!\rangle$, split in two parts. We write in grey the strengthened conditions obtained using $\langle\!\langle$cons$\rangle\!\rangle$, and $\underline{\text{underline}}$ the postcondition of the rule for the current atomic command. Everything else is a frame shared between pre and post, using $\langle\!\langle$frame$\rangle\!\rangle$.

convenient way to focus only on the part of the heap that describes the error, that is $x \not\mapsto$, and just leave everything else unspecified.

For the same program, in [Raa+20] the authors prove the ISL triple $[v \mapsto z * z \mapsto -]$ $\mathsf{rclient}$ $[v \mapsto y * y \mapsto - * x \not\mapsto\ ]$. This proves the existence of a faulty execution, but tells nothing about which input states actually lead to the error. On the other hand, the Separation SIL triple has a more succinct post capturing the error and exposes faulty initial states.

Lastly, we remark that, while in [ZDS23, Figure 6] outcome-based separation logic proves essentially the same triple, the deduction process is quite different from SIL. OL reasoning is forward oriented, as witnessed by the implication that concludes the proof and by the triple for the `skip` branch, whereas Separation SIL proof system naturally guides the backward inference.

5.4.6 Exit conditions in Separation SIL

We now briefly show how to adapt Separation SIL to handle different exit conditions. For this example, following [OHe20], we will consider ok and er, denoting correct and erroneous termination respectively. As discussed in Remark 2.9, this correspond to use $\{ok, er\} \times \Sigma$ as set of states instead of Σ_e. The denotational semantics of regular commands then acts as described for normal states (returning the error version of the current correct state instead of the generic **err**) and as the identity on error states.

The proof system changes accordingly: all the current rules are still valid with the ok flag (for atoms) or a generic flag ϵ (for structural rules) in both pre and postconditions. Rules introducing error flags are added for atoms. As an example, we write below the error rule for store.

$$\frac{}{\langle\!\langle ok : x \not\mapsto\ \rangle\!\rangle\ \texttt{[x] := y}\ \langle\!\langle er : x \not\mapsto\ \rangle\!\rangle}\ \langle\!\langle\text{store-er}\rangle\!\rangle$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle\!\langle ok : \texttt{len} \geq \texttt{cap} \land \texttt{len} > 7 \land (v \mapsto z * z \mapsto - * \mathbf{true})\rangle\!\rangle} \\
x := [v]; & \langle\!\langle ok : \texttt{len} \geq \texttt{cap} \land \texttt{len} > 7 \land (\mathbf{true} * v \mapsto z * z \mapsto - * (x = z \lor x \not\mapsto\ ))\rangle\!\rangle \\
(\texttt{len} \geq \texttt{cap})?; & \langle\!\langle ok : \texttt{len} > 7 \land (\mathbf{true} * v \mapsto z * z \mapsto - * (x = z \lor x \not\mapsto\ ))\rangle\!\rangle \\
y := [v]; & // \\
\mathrm{free}(y); & //\ \text{see Figure 5.9b} \\
y := \mathrm{alloc}(); & // \\
[v] := y; & \langle\!\langle ok : \texttt{len} + 1 > 8 \land (x \not\mapsto\ * \mathbf{true})\rangle\!\rangle \\
\texttt{len} := \texttt{len} + 1; & \langle\!\langle ok : \texttt{len} > 8 \land (x \not\mapsto\ * \mathbf{true})\rangle\!\rangle \\
\texttt{cap} := \texttt{cap} * 2; & \langle\!\langle ok : \texttt{len} > 8 \land (x \not\mapsto\ * \mathbf{true})\rangle\!\rangle \\
(\texttt{len} > 8)?; & \langle\!\langle ok : x \not\mapsto\ * \mathbf{true}\rangle\!\rangle \\
[x] := 10 & \langle\!\langle er : \mathbf{true}\rangle\!\rangle
\end{array}$$

Figure 5.10: Sketch of the derivation of the triple for rclient2 in Example 5.22. Postconditions to each statement are written on the right of the statement itself.

Moreover, the proof system is augmented with a rule for error propagation, that correspond to the fact that the semantics of programs on error states is the identity:

$$\frac{}{\langle\!\langle er : q\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle er : q\rangle\!\rangle}\ \langle\!\langle\mathsf{er\text{-}id}\rangle\!\rangle$$

The modified proof system is sound with respect to the modified semantics:

Theorem 5.21. *Any provable triple is valid:*

$$\vdash \langle\!\langle \epsilon : p\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle \epsilon' : q\rangle\!\rangle \implies \{\!|\epsilon : p|\!\} \subseteq [\![\overleftarrow{\mathsf{r}}]\!]\{\!|\epsilon' : q|\!\}$$

Example 5.22. Consider a refinement of the program in Example 5.20: here we assume that `len` and `cap` are two variables associated to the vector v describing its current length and capacity, respectively. We can then use them to decide the behavior of `push_back`: the vector gets reallocated only if the length after the insertion would exceed the current capacity. Moreover, we assume that `x` is used to access the element in position 8 of the vector, and therefore it's use after the `push_back` is guarded by a check that the vector is long enough. Therefore, the code becomes

$$\begin{array}{r}
\mathsf{rclient2} \triangleq x := [v];\ \text{if } (\texttt{len} >= \texttt{cap})\{\mathsf{r}_{b2}\} \text{ else } \{\texttt{len} := \texttt{len} + 1\}; \mathsf{r}_{use} \\
\mathsf{r}_{b2} \triangleq y := [v];\ \mathrm{free}(y);\ y := \mathrm{alloc}();\ [v] := y;\ \texttt{len} := \texttt{len} + 1;\ \texttt{cap} := \texttt{cap} * 2 \\
\mathsf{r}_{use} \triangleq \text{if } (\texttt{len} > 8)\{[x] := 10\}
\end{array}$$

To analyse this program, we use the error postcondition $\langle\!\langle er : \mathbf{true}\rangle\!\rangle$. For space constraints, we only consider the code path that goes through the then-branches of both if statements. The proof, linearized, is in Figure 5.10. The omitted part is analogous to the derivation in Figure 5.9b.

We highlight the use of rule $\langle\!\langle\mathsf{store\text{-}er}\rangle\!\rangle$ to infer the precondition $ok : x \not\mapsto\ * \mathbf{true}$ from the error postcondition $er : \mathbf{true}$ and $\langle\!\langle\mathsf{assume}\rangle\!\rangle$ to conjoin the boolean conditions in the preconditions of the guards. Moreover, the backward substitution of $\texttt{len} := \texttt{len} + 1$ performed by $\langle\!\langle\mathsf{assign}\rangle\!\rangle$ naturally propagates backward the constraint $\texttt{len} > 8$ to the value preceding the assignment, obtaining $\texttt{len} + 1 > 8$. This way, in the precondition of the whole command we explicitly find the constraints $\texttt{len} \geq \texttt{cap} \land \texttt{len} > 7$ on the initial values of `len` and `cap` that lead to the error. ■

5.4.7 Relative completeness of Separation SIL

The proof system in Figure 5.7 is complete for all atomic commands but `alloc`, because it misses the ability to refer the specific memory location that gets allocated. The naive solution to add $x = \alpha$ in the post of $\langle\!\langle \mathsf{alloc} \rangle\!\rangle$ would make the frame rule unsound: it would allow to prove, for instance, the invalid triple

$$\langle\!\langle \mathbf{emp} * \alpha \mapsto - \rangle\!\rangle \ \texttt{x := alloc()} \ \langle\!\langle (x \mapsto - \wedge x = \alpha) * \alpha \mapsto - \rangle\!\rangle.$$

Just like ISL needs the deallocated assertion in the post [Raa+20, §3], Separation SIL needs a "will be allocated" assertion in the pre. To this end, we use the $\not\mapsto$ assertion, and change the semantic model to only allocate a memory location that is explicitly δ. We formalize this by letting $avail(l) \triangleq h(l) = \delta$ in Figure 5.8a and replacing the axiom $\langle\!\langle \mathsf{alloc} \rangle\!\rangle$ with

$$\frac{}{\langle\!\langle \beta \not\mapsto \rangle\!\rangle \ \texttt{x := alloc()} \ \langle\!\langle x = \beta \wedge x \mapsto v \rangle\!\rangle} \ \langle\!\langle \mathsf{alloc} \rangle\!\rangle$$

The modified proof system is sound for this different semantics. Moreover, we can prove relative completeness [AO19, §4.3] for loop-free programs:

Theorem 5.23 (Relative completeness for loop-free programs)**.** *Suppose to have an oracle to prove implications between formulae in Asl. Let* $\mathsf{r} \in \mathsf{HRCmd}$ *be a regular command without* $\star$ *and* $p, q \in Asl$ *such that* $\llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!|q|\!\} \supseteq \{\!|p|\!\}$*. Then the triple* $\langle\!\langle p \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle q \rangle\!\rangle$ *is provable.*

The proof relies on an algorithmic rewriting of the postcondition that makes constraints on a single memory location explicit. This is given by the following lemma, whose proof is constructive (see the proof in Appendix C):

Lemma 5.24. *Let* $q \in Asl$ *be a formula without* $\exists$*, and let* x' *be a fresh variable. Then,*

1. *there exists a* $\bar{q}$ *such that* $q \wedge (x \mapsto x' * \textbf{\textit{true}}) \equiv x \mapsto x' * \bar{q}$
2. *there exists a* $\bar{q}$ *such that* $q \wedge (x \not\mapsto * \textbf{\textit{true}}) \equiv x \not\mapsto * \bar{q}$

Using this, we compute an assertion t whose semantics is precisely the weakest possible pre $\llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!|q|\!\}$ and prove the triple $\langle\!\langle t \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle q \rangle\!\rangle$. Then, using the oracle and $\langle\!\langle \mathsf{cons} \rangle\!\rangle$, the theorem follows for any p that implies t. Notably, this theorem shows that the weakest (possible) precondition $\llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!|q|\!\}$ of loop-free programs is always expressible as an assertion $t \in$ Asl, namely $\{\!|t|\!\} = \llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!|q|\!\}$, and that it can be computed algorithmically. This is result is far from trivial, as it depends on the expressiveness of the assertion language: for instance, if we add negation, the same result does not hold anymore because, even if more preconditions can be expressed, we can also introduce some new posts for which the precondition is not in the assertion language.

To extend the result to loops, we can focus on single states. Note that the following result does not need the oracle: it is always possible to craft a p whose proof only needs decidable implications.

Theorem 5.25 (State-wise completeness)**.** *Given a heap program* $\mathsf{r} \in \mathsf{HRCmd}$ *and two states* σ*,* σ' *such that* $\sigma \in \llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma'$*, for every assertion* q *such that* $\sigma' \in \{\!|q|\!\}$ *there exists an assertion* p *such that* $\sigma \in \{\!|p|\!\}$ *and* $\langle\!\langle p \rangle\!\rangle \ \mathsf{r} \ \langle\!\langle q \rangle\!\rangle$ *is provable in Separation SIL.*

5.4.8 Implementations

The ideas that lead to SIL represent *a posteriori* formalization and theoretical justification of the parallel, modular, and compositional static analysis implemented in industrial grade static analyzers for security developed and used at Meta, such as Zoncolan [DFLO19], Mariana Trench [Gab21], and Pysa [BC19]. Those tools automatically find more than 50% of the security bugs in the Meta family of apps and many SEVs ("severe bugs" in the Meta jargon) [DFLO19, Figure 5].

In order to scale up to hundreds of millions of lines of code, static analyses need to be parallelizable and henceforth modular and compositional. Modularity implies that the analysis can infer *meaningful* information without full knowledge of the global program. Compositionality means that the *results* of analyzing modules are good enough that one does not lose information in using the inferred triple instead of inlining the code.

The analysis implemented in the aforementioned tools is a modular backward analysis that determines which input states for a function will lead to a security error, likewise the SIL rules described in Figure 5.3. In particular, the analysis infers sufficient incorrectness preconditions (modularity) for callees that can be used by the callers (compositionality) to generate their incorrectness preconditions. When the propagation of the inferred sufficient precondition reaches an attacker-controlled input, the analysers check if that input is included in the propagated error condition. If it is the case, then it emits an error. The function analyses are parallelized and a strategy similar to the iteration rule of Figure 5.3 is used to compute the fixpoint in presence of mutually recursive functions.

Moreover, we have a proof-of-concept implementation of Separation SIL in OCaml. The prototype is open source and available on GitHub.[2] While the prototype is not meant to scale up to real programs, it can derive all the examples in this section and helped us gain confidence in the correctness of our results. The code exploits a routine that follows closely the relative completeness Theorem 5.23 by implementing Lemma 5.24. For this implementation, we thank the students of the Laboratory for Innovative Software 2024 course: Yuri Andriaccio, Samuele Bonini, Andrea Castagna, Marco Antonio Corallo, Andrea Simone Costa, Fabio Federico, Elvis Rossi, Alessandro Scala, Matteo Simone.

5.5 Conclusions

In this chapter, we considered known program logics for over and under-approximation and tried to formalize the relations among them in order to asses their respective strengths and weaknesses. This led to the introduction of the novel proof system for Lisbon triples, that we dubbed Sufficient Incorrectness Logic, and to compare the four logics along several dimensions. We also instantiated the new SIL proof system to handle pointers and memory allocation. The resulting Separation SIL is able to identify the causes of memory errors. We presented a first, simpler version of Separation SIL for which we provide a prototype implementation. We also showed how to manage explicit errors via ok/er flags, and how to modify the logic to become complete. Comparing the logics, we recovered some known results as well as finding new connections and highlighting insights that shed a little more light on the asymmetries between over/under-approximation an forward/backward analysis. The shortest summary of our findings is that there is no silver bullet: each logic has its own strengths and use cases.

[2] `https://github.com/Alex23087/Failure-SSIL-Analyser`

Chapter 6

Local Completeness Logic

In this chapter, we extend LCL_A (see Section 3.2) with new capabilities. First, we investigate the possibility of relaxing point (3) of Theorem 3.7 to $[\![\mathsf{r}]\!]^A\alpha(P) = \alpha(Q)$ to achieve extensional soundness, i.e., to untie the set of properties that can be proved about the function computed by the program from the way the program is written. To do so, we follow the idea introduced in [BGGR23, §8] of *changing the abstract domain* during the analysis, possibly in different ways for different portions of the code. While [BGGR23] proposes a single rule for domain refinement, we study here both *refinement* and *simplification* rules for LCL_A. Moreover, we study here how to weaken the hypothesis of Galois connection: the whole theory of completeness is based on the existence of a best approximation for concrete points, but this is not always available in practical instances [CC92]. Lastly, we study further the possibility of using LCL_A for backward analysis, as briefly outlined in [BGGR23, §5.3].

The content of this chapter is based on [ABG23] and [ABGL24, §6].

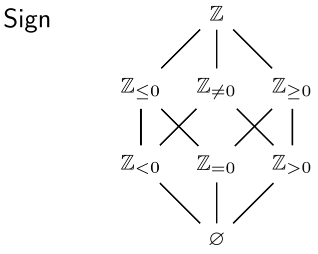
6.1 Motivation

While any LCL_A valid triple allows to prove both correctness and incorrectness, a very powerful ability, proving an LCL_A triple can be challenging. Therefore, it is important to study how to simplify this task.

The strongest of the three properties required by LCL_A is point (3) in Theorem 3.7, which in turn is key to guarantee that point (2) holds. However, requiring the abstract interpreter $[\![\mathsf{c}]\!]^\sharp_A$ to be locally complete is an *intensional* property, because the function $[\![\mathsf{c}]\!]^\sharp_A$ depends on how c is written. In fact, it is well known that the abstract analyses $[\![\mathsf{c}_1]\!]^\sharp_A$ and $[\![\mathsf{c}_2]\!]^\sharp_A$ of two programs c_1 and c_2 computing the same function (i.e., $[\![\mathsf{c}_1]\!] = [\![\mathsf{c}_2]\!]$) can yield different results.

A weaker requirement that suffices to guarantee the validity of point (2) is the local completeness of the bca $[\![\mathsf{c}]\!]^A$ of the function $[\![\mathsf{c}]\!]$, which is an *extensional* property: it only depends on the abstract domain A and on the computed function $[\![\mathsf{c}]\!]$ associated with c, not on the way c is composed. In its original formulation, LCL_A exploits $[\![\mathsf{c}]\!]^\sharp_A$ instead of $[\![\mathsf{c}]\!]^A$ because the second can be as hard to compute as the concrete semantics $[\![\mathsf{c}]\!]$ and because the sequential composition of bcas is not necessarily a bca itself. The difference between $[\![\mathsf{c}]\!]^A$ and $[\![\mathsf{c}]\!]^\sharp_A$ is exemplified below (see also, e.g., [LL09, Example 1]):

Example 6.1 (Extensional and intensional properties). Consider the concrete domain $\mathcal{P}(\mathbb{Z})$ of sets of integers and the abstract domain of signs given below:



The meaning of each abstract elements of Sign is to represent concrete values that satisfy the respective property: for instance, $\gamma(\mathbb{Z}_{<0}) = \{n \in \mathbb{Z} \mid n < 0\}$ and $\alpha(\{0;1;100\}) = \mathbb{Z}_{\geq 0}$. The bca of a concrete function $f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ is defined as $f^{\mathsf{Sign}} \triangleq \alpha\, f\, \gamma : \mathsf{Sign} \to \mathsf{Sign}$.

Consider the simple program fragment

$$\mathsf{c} \triangleq \texttt{x := x + 1; x := x - 1} \,.$$

It's denotational semantics $[\![\mathsf{c}]\!]$ is the identity $\mathrm{id}_{\mathbb{Z}}$, so its bca (see Definition 2.19)

$$[\![\mathsf{c}]\!]^{\mathsf{Sign}} \triangleq \alpha\ \mathrm{id}_{\mathbb{Z}}\ \gamma = \mathrm{id}_{\mathsf{Sign}}$$

is just the abstract identity. We say that $[\![\mathsf{c}]\!]^{\mathsf{Sign}}$ is *extensional* because it only depends on the function computed by c, i.e., its denotational semantics. However, an analyser does not know the semantics of c, so it has to analyse the program syntactically, breaking it down into elementary pieces and gluing the results together. For instance, starting from the concrete point $P = \{1\}$, the analysis first abstracts it to the property $\alpha(P) = \mathbb{Z}_{>0}$, then it computes

$$\begin{aligned} [\![\mathsf{c}]\!]^{\sharp}_{\mathsf{Sign}}(\mathbb{Z}_{>0}) &= [\![\texttt{x := x - 1}]\!]^{\sharp}_{\mathsf{Sign}} [\![\texttt{x := x + 1}]\!]^{\sharp}_{\mathsf{Sign}}(\mathbb{Z}_{>0}) \\ &= [\![\texttt{x := x - 1}]\!]^{\sharp}_{\mathsf{Sign}}(\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}. \end{aligned}$$

Analogous calculations for all properties in Sign yields the abstraction

$$[\![\mathsf{c}]\!]^{\sharp}_{\mathsf{Sign}}(a) = \begin{cases} \varnothing & \text{if } a = \varnothing \\ \mathbb{Z}_{\geq 0} & \text{if } a \in \{\, \mathbb{Z}_{=0} \,,\, \mathbb{Z}_{>0} \,, \mathbb{Z}_{\geq 0}\} \\ \mathbb{Z}_{<0} & \text{if } a = \mathbb{Z}_{<0} \\ \mathbb{Z} & \text{if } a \in \{\, \mathbb{Z}_{\leq 0} \,,\, \mathbb{Z}_{\neq 0} \,, \mathbb{Z}\} \end{cases}$$

that, albeit sound, is less precise than $\mathrm{id}_{\mathsf{Sign}}$ (we highlight with a gray background all inputs on which $[\![\mathsf{c}]\!]^{\sharp}_{\mathsf{Sign}}$ loses accuracy).

The program c is equivalent to the command $\texttt{skip}$, because $[\![\texttt{skip}]\!] = \mathrm{id}_{\mathbb{Z}}$, and thus c and $\texttt{skip}$ are assigned the same bca $[\![\texttt{skip}]\!]^{\mathsf{Sign}} = [\![\mathsf{c}]\!]^{\mathsf{Sign}} = \mathrm{id}_{\mathsf{Sign}}$. However, $[\![\texttt{skip}]\!]^{\sharp}_{\mathsf{Sign}} = \mathrm{id}_{\mathsf{Sign}} \neq [\![\mathsf{c}]\!]^{\sharp}_{\mathsf{Sign}}$, exposing the *intensional* essence of the abstract interpreter: the abstraction depends on how the program is written and not only on its semantics.[1] ■

The possibility of weakening point (3) from being an intensional requirement based on $[\![\mathsf{c}]\!]^{\sharp}_{A}$ to an extensional one based on bcas $[\![\mathsf{c}]\!]^{A}$ has several nice consequences. First, the local completeness of $[\![\mathsf{c}]\!]^{A}$ is enough to guarantee that points (1-2) hold. Second, while the proof system itself provides an intensional analysis, because its rules are defined inductively on the program syntax, the information it produces is more precise, in the

[1] While it falls outside the scope of this thesis, we refer the interested reader to, e.g., [Bru+19; BRZ22] for more about intensional and extensional abstract properties.

sense that the property associated with triples is extensional: no precision is lost because of the approximation introduced by the intensional abstract interpreter. Third, it allows the proof system to derive more triples than the original one because a bca can be locally complete even when the abstract interpreter is not (but not vice versa). Finally, since extensional properties are independent of how the program is written, this possibility provides the potential for deriving exactly the same triples for equivalent programs.

Another constraint imposed by LCL_A is the need for a Galois connection: the whole theory of completeness in abstract interpretation is based on it. Therefore, LCL_A cannot be applied to known instances of abstract domains lacking an abstraction function α, such as convex polyhedra [CH78] that are widely used in static analysis. However, abstract convexity of local completeness can help mitigate this limitation: even if a point doesn't have a best abstraction, if it can be bound between another point and its abstraction, we can in a sense "prove" local completeness on it thanks to abstract convexity.

Lastly, LCL_A was proposed for forward analysis. In theory, nothing prevents it to be used in a backward fashion, but the classical forward/backward duality requires the use of under-approximation abstract domains, making it impractical (see Chapter 4). However, if we consider the backward semantics $\llbracket \overleftarrow{\cdot} \rrbracket$ defined in the previous chapter, it turns out that it is possible to combine SIL (Section 5.2) with over-approximation abstract domain in a LCL_A-style.

6.2 Extensional soundness

As anticipated at the beginning of the chapter, one of our goals is to weaken point (3) of the soundness Theorem 3.7 from local completeness of the abstract interpreter $\llbracket \mathsf{r} \rrbracket^\sharp_A$ to that of the bca $\llbracket \mathsf{r} \rrbracket^A$. The key observation is that the proof of Corollary 3.8 only relies on points (1-2) of Theorem 3.7, so from a program analysis perspective point (3) can be seen as a technical requirement to prove the other two. However, we observe that local completeness of the bca is enough to this aim: we therefore present a slightly weaker[2] soundness result for LCL_A, with the bca $\llbracket \mathsf{r} \rrbracket^A$ in place of the inductively defined abstract interpreter $\llbracket \mathsf{r} \rrbracket^\sharp_A$.

Theorem 6.2 (Extensional soundness)**.** *Let $A_{\alpha,\gamma} \in \mathrm{Abs}(C)$. If $\vdash_A [P]\ \mathsf{r}\ [Q]$ then:*

1. *$Q \leq \llbracket \mathsf{r} \rrbracket P$,*
2. *$\alpha(\llbracket \mathsf{r} \rrbracket P) = \alpha(Q)$,*
3. *$\llbracket r \rrbracket^A \alpha(P) = \alpha(Q)$.*

Proof. First we remark that points (1) and (3) implies point (2):

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && [(1) \text{ and monotonocity of } \alpha] \\
&\leq \llbracket \mathsf{r} \rrbracket^A \alpha(P) && [\text{soundness of } \llbracket \mathsf{r} \rrbracket^A] \\
&= \alpha(Q) && [(3)]
\end{aligned}$$

So all the lines are equal, in particular $\alpha(Q) = \alpha(\llbracket \mathsf{r} \rrbracket P)$. The proof is then by induction on the derivation tree of $\vdash_A [P]\ \mathsf{r}\ [Q]$, but we only have to prove (1) and (3) because of the observation above. We only include one inductive case as an example; the full proof is in Appendix D.

[2] Logically speaking, we prove a stronger conclusion, so the theorem as an implication is weaker.

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)}{\vdash_{A} [P]\ \mathsf{r}\ [Q]}\ (\mathsf{refine\text{-}ext})$$

Figure 6.1: Rule (refine-ext) for LCL$_A$.

Case (seq)
(1) $Q \leq [\![\mathsf{r}_2]\!]R \leq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!]P) = [\![\mathsf{r1}; \mathsf{r}_2]\!]P$, where the inequalities follow from inductive hypotheses and monotonicity of $[\![\mathsf{r}_2]\!]$.
(3) We recall that $[\![\mathsf{r}_1; \mathsf{r}_2]\!]^A \leq [\![\mathsf{r}_2]\!]^A[\![\mathsf{r}_1]\!]^A$.

$$\begin{aligned}
\alpha(Q) &\leq \alpha([\![\mathsf{r}_1; \mathsf{r}_2]\!]P) && [(1) \text{ and monotonicity of } \alpha] \\
&\leq [\![\mathsf{r}_1; \mathsf{r}_2]\!]^A\alpha(P) && [\text{soundness of } [\![\mathsf{r}]\!]^A] \\
&\leq [\![\mathsf{r}_2]\!]^A[\![\mathsf{r}_1]\!]^A\alpha(P) && [\text{recalled above}] \\
&= [\![\mathsf{r}_2]\!]^A\alpha(R) && [\text{inductive hp}] \\
&= \alpha(Q) && [\text{inductive hp}]
\end{aligned}$$

So all the lines are equal, in particular $[\![\mathsf{r}_1; \mathsf{r}_2]\!]^A\alpha(P) = \alpha(Q)$. □

Theorem 3.7 involves $[\![\mathsf{r}]\!]^\sharp_A$, an *intensional* property of the program r that depends on how the program is written, while the new statement we propose here relies only on $[\![\mathsf{r}]\!]^A$, an *extensional* property of the computed function $[\![\mathsf{r}]\!]$ and not of r itself. Accordingly, for the rest of this chapter we use the name *intensional soundness* for the former and *extensional soundness* for the latter. Again, we say a triple is *extensionally valid* if it satisfies point (1–3) of Theorem 6.2 above, and intensionally valid for the former notion introduced in Section 3.2. We shall write $\vDash_A [P]\ \mathsf{r}\ [Q]$ for both, but we will make sure to disambiguate the notation when not clear from the context.

6.3 Locally complete refinement

Our aim is to enhance the original LCL$_A$ proof system to handle triples where the extensional abstraction $[\![\mathsf{r}]\!]^A$ is proved to be locally complete w.r.t. the given input, that is $[\![\mathsf{r}]\!]^A\alpha(P) = \alpha([\![\mathsf{r}]\!]P)$. To achieve this, we extend the proof system with a new inference rule, that is shown in Figure 6.1. It is named after "refine" because it allows to refine the abstract domain A to some $A' \preceq A$ (see Section 2.9) and "ext" since it involves the extensional bca $[\![\mathsf{r}]\!]^{A'}$ of $[\![\mathsf{r}]\!]$ in A' (to distinguish it from the rules we will introduce later).

Using (refine-ext) it is possible to construct a derivation that proves local completeness of portions of the whole program in a more precise abstract domain A' and then carries the result over to the global analysis in a coarser domain A. The only requirement for the application of the rule is that the domain A' satisfies $A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)$.

Formally, given the two abstract domains $A_{\alpha,\gamma}, A'_{\alpha',\gamma'} \in \mathrm{Abs}(C)$, this last premise of rule (refine-ext) should be written as $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) = \alpha(Q)$. However we prefer the more concise, albeit a little imprecise, notation in Figure 6.1. That notation is justified by the following intuitive argument: since $A' \preceq A$ we can consider, with a slight abuse of notation (seeing abstract domains as closures) $A \subseteq A' \subseteq C$, so that for any element $a \in A \subseteq C$ we have $\gamma(a) = \gamma'(a) = a$ and for any $c \in C$ we have $\alpha'A(c) = A(c)$. With these

$$\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) = \alpha[\![\mathsf{r}]\!]^{A'}A(P) = A[\![\mathsf{r}]\!]^{A'}A(P).$$

With the addition of rule (refine-ext), intensional soundness (Theorem 3.7) does not hold anymore: since this rule allows to perform part of the analysis in a more concrete domain A', we do not get any information on $[\![\mathsf{r}]\!]^{\sharp}_{A}$. However, rule (refine-ext) is sound w.r.t. the bca $[\![\mathsf{r}]\!]^{A}$, and therefore it makes the proof system extensionally sound:

Theorem 6.3 (Extensional soundness of (refine-ext))**.** *The proof system in Figure 3.2 with the addition of rule* (refine-ext) *from Figure 6.1 is extensionally sound.*

Proof sketch. We extend the proof of Theorem 6.2 with a new inductive case. The full details are in Appendix D. □

We remark that a rule like (refine-ext), that allows to carry on part of the proof in a different abstract domain, cannot be unconstrained. We present an example showing that an analogous inference rule only requiring the triple $\vdash [P]\ \mathsf{r}\ [Q]$ to be provable in an abstract domain $A' \preceq A$ without any further constraint would be unsound.

Example 6.4. Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the point $P = \{-5; -1\}$, the abstract domain Sign of Example 6.1 and the program

$$\mathsf{r} \triangleq \texttt{x := x + 10}.$$

Then $C \preceq \mathsf{Sign}$ and we can prove $\vdash_C [P]\ \mathsf{r}\ [\{5; 9\}]$ applying (transfer) since all functions are locally complete in the concrete domain. However, if $f = [\![\mathsf{r}]\!] = (\!|\texttt{x := x + 10}|\!)$, it is not the case that $\mathbb{C}^{\mathsf{Sign}}_{P}(f)$: indeed

$$\mathsf{Sign}(f(\mathsf{Sign}(P))) = \mathsf{Sign}(f(\mathbb{Z}_{<0})) = \mathsf{Sign}(\{n \in \mathbb{Z} \mid n < 10\}) = \top$$

while

$$\mathsf{Sign}(f(P)) = \mathsf{Sign}(\{5, 9\}) = \mathbb{Z}_{>0}.$$

This means that a refinement rule without any additional condition is unsound because it would allow to prove triples which are not locally complete. ■

6.3.1 Logical completeness

Among all the possible conditions that make a refinement rule valid, we believe ours to be very general since (refine-ext) allows us to derive logical completeness, that is, the ability to prove *any* triple satisfying the soundness properties guaranteed by the proof system. Note that this was not the case for the original LCL$_A$ proof system [BGGR21, §5.2].

However, to prove such a result, our extension need an additional rule to handle loops, just like the original LCL$_A$ and other logics (IL, SIL). The necessary infinitary rule, called (limit), allows the proof system to handle Kleene star, and is the same as LCL$_A$:

$$\frac{\forall n \geq 0 \mid\ \vdash_A [P_n]\ \mathsf{r}\ [P_{n+1}]}{\vdash_A [P_0]\ \mathsf{r}^{\star}\ [\bigvee_{i \geq 0} P_i]}\ (\mathsf{limit})$$

Theorem 6.5 (Logical completeness of (refine-ext))**.** *Consider the proof system of Figure 3.2 with the addition of rules* (refine-ext) *and* (limit)*. If $Q \leq [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^{A}\alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

Proof. First, the hypotheses of the theorem implies $\mathbb{C}^A_P(\llbracket \mathsf{r} \rrbracket)$:

$$\begin{aligned} \llbracket \mathsf{r} \rrbracket^A \alpha(P) &= \alpha(Q) && \text{[hp of the theorem]} \\ &\le \alpha(\llbracket \mathsf{r} \rrbracket P) && \text{[monotonicity of } \alpha \text{ and hp } Q \le \llbracket \mathsf{r} \rrbracket P] \\ &\le \llbracket \mathsf{r} \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A] \end{aligned}$$

Hence $\alpha(\llbracket \mathsf{r} \rrbracket P) = \llbracket \mathsf{r} \rrbracket^A \alpha(P) = \alpha \llbracket \mathsf{r} \rrbracket \gamma \alpha(P)$, that is local completeness, and $\alpha(Q) = \alpha(\llbracket \mathsf{r} \rrbracket P)$.

Now consider r, P, Q satisfying the hypotheses. If $Q < \llbracket \mathsf{r} \rrbracket P$, by (relax) we get

$$\frac{P \le P \le A(P) \quad \vdash_A [P] \; \mathsf{r} \; [\llbracket \mathsf{r} \rrbracket P] \quad Q \le \llbracket \mathsf{r} \rrbracket P \le A(Q)}{\vdash_A [P] \; \mathsf{r} \; [Q]} \text{(relax)}$$

But the first condition is trivial, and the third one is made of $Q \le \llbracket r \rrbracket P$ (the hypothesis) and $\llbracket r \rrbracket P \le A(Q)$, that follows because $\alpha(\llbracket r \rrbracket P) = \alpha(Q)$ (shown above) and in a Galois connection this implies $\llbracket r \rrbracket P \le \gamma \alpha(Q) = A(Q)$. Hence, without loss of generality, we can assume $Q = \llbracket \mathsf{r} \rrbracket P$.

Now we want to apply (refine-ext) to move to the concrete domain C. Clearly $C \preceq A$. The last hypothesis of the rule can be readily verified recalling that $\llbracket r \rrbracket^C = \llbracket r \rrbracket$ and $\alpha' = \gamma' = \mathrm{id}_C$:

$$\begin{aligned} \alpha \llbracket r \rrbracket^C A(P) &= \alpha \llbracket r \rrbracket A(P) \\ &= \llbracket r \rrbracket^A \alpha(P) \\ &= \alpha(\llbracket \mathsf{r} \rrbracket P) \end{aligned}$$

To say that triple $\vdash_C [P] \; \mathsf{r} \; [\llbracket \mathsf{r} \rrbracket P]$ is provable we resort to Theorem 5.11 of [BGGR21]. The hypothesis of that theorem are satisfied: all expressions are globally complete in the concrete domain C, $\llbracket r \rrbracket P \le \llbracket r \rrbracket P$ and $\llbracket r \rrbracket^\sharp_C \mathrm{id}_C(P) = \llbracket \mathsf{r} \rrbracket P = \mathrm{id}_C(\llbracket r \rrbracket P)$, where we used $\alpha' = \mathrm{id}_C$ and $\llbracket r \rrbracket^\sharp_C = \llbracket r \rrbracket$.

Thus, by applying (refine-ext), we can prove the triple $\vdash_A [P] \; \mathsf{r} \; [\llbracket \mathsf{r} \rrbracket P]$:

$$\frac{\vdash_C [P] \; \mathsf{r} \; [\llbracket \mathsf{r} \rrbracket P] \quad C \preceq A \quad A \llbracket \mathsf{r} \rrbracket^C A(P) = A(\llbracket \mathsf{r} \rrbracket P)}{\vdash_A [P] \; \mathsf{r} \; [\llbracket \mathsf{r} \rrbracket P]} \text{(refine-ext)}$$

□

The previous theorem proves the logical completeness of our proof system with respect to extensional validity: indeed, if $Q \le \llbracket \mathsf{r} \rrbracket P$ and $\llbracket \mathsf{r} \rrbracket^A \alpha(P) = \alpha(Q)$ we also have $\alpha(\llbracket \mathsf{r} \rrbracket P) = \alpha(Q)$ (see, e.g., the proof of Theorem 6.2).

An interesting consequence of this result is the existence of a refinement A' in which it is possible to carry out the proof. In principle, such a refinement could be the concrete domain C (as shown in the proof), that is not computable. However, it is worth nothing that for a sequential fragment (a portion of code without loops) the concrete domain can be actually used (for instance via first-order logic). This opens up the possibility, for instance, to infer a loop invariant on the body using C, and then prove it using an abstract domain. In Section 6.3.3 we discuss this issue further.

6.3.2 Derived refinement rules

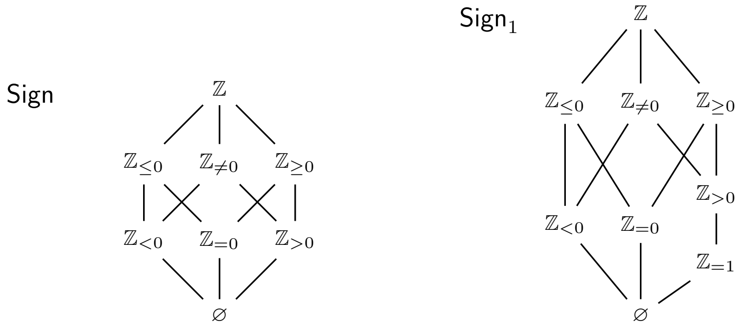
The hypothesis $A \llbracket \mathsf{r} \rrbracket^{A'} A(P) = A(Q)$ is added to rule (refine-ext) in order to guarantee soundness: in general, the ability to prove a triple such as $\vdash [P] \; \mathsf{r} \; [Q]$ in a refined domain A' only gives information on $A \llbracket \mathsf{r} \rrbracket^{A'} A'(P)$ but not on $A \llbracket \mathsf{r} \rrbracket^{A'} A(P)$. In fact, the example below shows that $A \llbracket \mathsf{r} \rrbracket^{A'} A'(P)$ and $A \llbracket \mathsf{r} \rrbracket^{A'} A(P)$ can be different.

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A[\![\mathsf{r}]\!]^{\sharp}_{A'}A(P) = A(Q)}{\vdash_{A} [P]\ \mathsf{r}\ [Q]}\ (\textsf{refine-int})$$

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \preceq A \quad A'(P) = A(P)}{\vdash_{A} [P]\ \mathsf{r}\ [Q]}\ (\textsf{refine-pre})$$

Figure 6.2: Derived refinement rules for LCL_A.

Example 6.6. Consider the concrete domain $\mathcal{P}(\mathbb{Z})$, the abstract domain of signs $\mathsf{Sign}_{\alpha,\gamma} \in \mathrm{Abs}(\mathcal{P}(\mathbb{Z}))$ (introduced in Example 6.1) and its refinement Sign_1 below:



For the command $\mathsf{r} \triangleq$ `x := x - 1` and the concrete point $P = \{1\}$ we have

$$\mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}\mathsf{Sign}_1(P) = \mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}(\mathbb{Z}_{=1}) = \mathbb{Z}_{=0}$$

but

$$\mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}\mathsf{Sign}(P) = \mathsf{Sign}[\![\mathsf{r}]\!]^{\mathsf{Sign}_1}(\mathbb{Z}_{>0}) = \mathbb{Z}_{\geq 0}.$$

■

Despite being necessary, the hypothesis of rule (refine-ext) cannot be checked algorithmically because, in general, the bca $[\![\mathsf{r}]\!]^{A'}$ of a composite command r is not computable. To mitigate this issue, we present in Figure 6.2 two derived rules whose premises imply the premises of (refine-ext), thus ensuring extensional soundness via Theorem 6.3.

The first rule we present replaces the requirement on the extensional bca $[\![\mathsf{r}]\!]^{A'}$ with requirements on the intensional compositional abstraction $[\![\mathsf{r}]\!]^{\sharp}_{A'}$ computed in A'. For this reason, we call this rule (refine-int).

The second derived rule we propose is simpler than (refine-ext): it just requires the abstractions $A(P)$ and $A'(P)$ to coincide, with no reference to the regular command r nor to the postcondition Q. Since the premise is only on the precondition P, we call this rule (refine-pre).

Proposition 6.7. *Both rules* (refine-int) *and* (refine-pre) *in Figure 6.2 are extensionally sound.*

Proof sketch. We show that the hypotheses of both rules imply those of (refine-ext). Since the first two hypotheses $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ and $A' \preceq A$ are shared among the rules, we only have to show that $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha' A(P) = \alpha(Q)$. The details are in Appendix D.

Because the hypotheses of (refine-int) and (refine-pre) implies those of (refine-ext), whenever we can apply the former we can also apply the latter, so that Theorem 6.3 ensures extensional soundness. □

It is worth noting that the condition in (refine-int) on the compositional abstraction $[\![\mathsf{r}]\!]^\sharp_{A'}$ can easily be checked by the analyser, possibly alongside the analysis of r with LCL or using a stand-alone abstract interpreter. Moreover, this rule is as powerful as the original (refine-ext) because it enjoys a logical completeness result akin to Theorem 6.5.

Theorem 6.8 (Logical completeness of (refine-int))**.** *Consider the proof system of Figure 3.2 with the addition of rules* (refine-int) *and* (limit)*. If $Q \leq [\![\mathsf{r}]\!]P$ and $[\![\mathsf{r}]\!]^A\alpha(P) = \alpha(Q)$ then $\vdash_A [P]\ \mathsf{r}\ [Q]$.*

Proof. The proof is the same as Theorem 6.5, the only difference being that to apply (refine-int) we need to show $A[\![\mathsf{r}]\!]^\sharp_C A(P) = A([\![\mathsf{r}]\!]P)$ instead of $A[\![\mathsf{r}]\!]^C A(P) = A([\![\mathsf{r}]\!]P)$. However, since in the concrete domain $[\![\mathsf{r}]\!]^\sharp_C = [\![\mathsf{r}]\!]^C = [\![\mathsf{r}]\!]$ the proof still holds. □

Just like logical completeness of (refine-ext), this result implies the existence of a refinement A' (possibly the concrete domain) in which it is possible to carry out the proof.

Rule (refine-pre) only requires a simple check at the application site instead of an expensive analysis of the program r, so it can be preferable in practice. We present an example to highlight the advantages of this rule (as well as (refine-int)), which allows us to use different domains in the proof derivation of different parts of the program.

Example 6.9 (The use of (refine-pre)). Consider the two program fragments

$$\begin{aligned}
\mathsf{r}_1 &\triangleq \texttt{(y != 0)?; y := abs(y)} \\
\mathsf{r}_2 &\triangleq \texttt{x := y; while (x > 1) \{ y := y - 1; x := x - 1 \}} \\
&= \texttt{x := y; ((x > 1)?; y := y - 1; x := x - 1)}^\star\texttt{; (x <= 1)?}
\end{aligned}$$

and the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Here abs is a function to compute the absolute value, and we assume, for the sake of simplicity, that the analyser knows its best abstraction. Consider the concrete domain $\mathcal{P}(\mathbb{Z}^2)$ where a pair (n, m) denote a state $\mathtt{x} = n$, $\mathtt{y} = m$, and the initial state $P = (\mathtt{y} \in [-100; 100])$, a logical description of the concrete points $\{(n, m) \mid m \in [-100; 100]\} \in \mathcal{P}(\mathbb{Z}^2)$. The bca $[\![\mathsf{r}]\!]^{\text{Int}}$ in the abstract domain of intervals is locally complete on P (since P is expressible in Int), but the compositional abstraction $[\![\mathsf{r}]\!]^\sharp_{\text{Int}}$ is not:

$$\begin{aligned}
[\![\mathsf{r}]\!]^{\text{Int}}\alpha(P) &= \text{Int}([\![\mathsf{r}_2]\!][\![\mathsf{r}_1]\!](\{(n, m) \mid m \in [-100; 100]\})) \\
&= \text{Int}([\![\mathsf{r}_2]\!](\{(n, m) \mid m \in [1; 100]\})) \\
&= \text{Int}(\{(1, 1)\}) \\
&= ([1; 1] \times [1; 1]),
\end{aligned}$$

while

$$\begin{aligned}
[\![\mathsf{r}]\!]^\sharp_{\text{Int}}\alpha(P) &= [\![\mathsf{r}_2]\!]^\sharp_{\text{Int}}[\![\mathsf{r}_1]\!]^\sharp_{\text{Int}}([-\infty; +\infty] \times [-100; 100]) \\
&= [\![\mathsf{r}_2]\!]^\sharp_{\text{Int}}[\![\texttt{y := abs(y)}]\!]^{\text{Int}}([-\infty; +\infty] \times [-100; 100]) \\
&= [\![\mathsf{r}_2]\!]^\sharp_{\text{Int}}([-\infty; +\infty] \times [0; 100]) \\
&= ([1; 1] \times [0; 100]) \neq ([1; 1] \times [1; 1]).
\end{aligned}$$

The issues are twofold. First, the analysis of r_1 in Int is incomplete, so we need a more concrete domain. For instance, we can select $\text{Int}_{\neq 0}$, the Moore closure of Int with the addition of the element $\mathbb{Z}_{\neq 0}$ representing the property of being nonzero. Intuitively, $\text{Int}_{\neq 0}$ contains all intervals, possibly having a "hole" in 0. Formally

$$\text{Int}_{\neq 0} = \text{Int} \cup \{I_{\neq 0} \mid I \in \text{Int}\}$$

with $\gamma'(I_{\neq 0}) = \gamma(I) \setminus \{0\}$.

While there is no need for a relational domain to analyse r_1 since variable `x` is never mentioned in it, the analysis of r_2 requires a relational domain to track the information that the value of variable `x` is equal to the value of variable `y`. This suggests to use the octagons domain Oct [Min06] to analyse r_2. It is worth noting that the abstract domain Oct would not be able to perform a locally complete analysis of r_1 for the same reasons that the domain Int could not.

However, rule (refine-pre) allows us to combine these different proof derivations. Since the program state between r_1 and r_2 can be precisely represented in Int, we use this domain as a baseline and refine it to $\mathrm{Int}_{\neq 0}$ and to Oct for the two parts, respectively.

Let $R = (\mathtt{y} \in \{1; 2; 100\})$ that is an under-approximation of the concrete state in between r_1 and r_2 with the same abstraction in Int, so the triple $\vdash_{\mathrm{Int}} [P]\ \mathsf{r}_1\ [R]$ is valid. Note that the concrete point 2 was added to R in order to have local completeness for `(x > 1)?` in r_2. However, this triple cannot be proved in Int because $\llbracket \mathsf{r}_1 \rrbracket^\sharp_{\mathrm{Int}}$ is not locally complete on P, so we resort to (refine-pre) to change the domain to $\mathrm{Int}_{\neq 0}$. In the derivation below, we let $R_1 = (\mathtt{y} \in [-100; 100] \wedge \mathtt{y} \neq 0)$ and we omit for simplicity the additional hypothesis of (relax):

$$\dfrac{\dfrac{\mathbb{C}^{\mathrm{Int}_{\neq 0}}_{P}(\llbracket \texttt{y != 0?} \rrbracket)}{\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \texttt{y != 0?}\ [R_1]}\ (\mathsf{transfer}) \qquad \dfrac{\dfrac{\mathbb{C}^{\mathrm{Int}_{\neq 0}}_{R_1}(\llbracket \texttt{y := abs(y)} \rrbracket)}{\vdash_{\mathrm{Int}_{\neq 0}} [R_1]\ \texttt{y := abs(y)}\ [\mathtt{y} \in [1; 100]]}\ (\mathsf{transfer})}{\vdash_{\mathrm{Int}_{\neq 0}} [R_1]\ \texttt{y := abs(y)}\ [R]}\ (\mathsf{relax})}{\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]}\ (\mathsf{seq})$$

Again $\llbracket \mathsf{r}_2 \rrbracket$ is locally complete on R in Int, but the compositional analysis $\llbracket \mathsf{r}_2 \rrbracket^\sharp_{\mathrm{Int}}$ is not. Hence to perform the derivation we resort to (refine-pre) to introduce relational information in the abstract domain, using Oct instead of Int. Let $Q = (\mathtt{x} = 1 \wedge \mathtt{y} = 1)$, that is the concrete output of the program, so that we can prove $\vdash_{\mathrm{Int}} [R]\ \mathsf{r}_2\ [Q]$. The derivation of this triple is in Appendix D, Figure D.1. However, the proof is just a straightforward application of rules (seq), (iterate) and (transfer).

With those two derivation, we can prove the triple $\vdash_{\mathrm{Int}} [P]\ \mathsf{r}\ [Q]$ using (refine-pre):

$$\dfrac{\dfrac{\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}_1\ [R]}{\vdash_{\mathrm{Int}} [P]\ \mathsf{r}_1\ [R]}\ (\mathsf{refine\text{-}pre}) \qquad \dfrac{\vdash_{\mathrm{Oct}} [R]\ \mathsf{r}_2\ [Q]}{\vdash_{\mathrm{Int}} [R]\ \mathsf{r}_2\ [Q]}\ (\mathsf{refine\text{-}pre})}{\vdash_{\mathrm{Int}} [P]\ \mathsf{r}\ [Q]}\ (\mathsf{seq})$$

For space constraints, we write here the additional hypotheses of the rules. For the first application, $\mathrm{Int}_{\neq 0} \preceq \mathrm{Int}$ and $\mathrm{Int}_{\neq 0}(P) = P = \mathrm{Int}(P)$. For the second, $\mathrm{Oct} \preceq \mathrm{Int}$ and $\mathrm{Int}(R) = (\mathtt{y} \in [1; 100]) = \mathrm{Oct}(R)$.

It is worth noting that, in this example, all applications of (refine-pre) can be replaced by (refine-int). This means that also the latter is able to exploit $\mathrm{Int}_{\neq 0}$ and Oct to prove the triple in the very same way, but its application requires more expensive abstract analyses than the simple checks of (refine-pre). ■

While (refine-pre) is simpler than (refine-ext) and (refine-int), it is also weaker in both a theoretical and practical sense. On the one hand, LCL_A extended with this rule does not admit a logical completeness result; on the other hand, there are situations in which, even though (refine-pre) allows a derivation, (refine-int) is more effective. We show these two points by examples. For the first, we propose a valid triple that LCL_A extended with (refine-pre) cannot prove.

Example 6.10 (Logical incompleteness of (refine-pre)). Consider the program fragments[3]

$$\begin{aligned}\mathsf{r}_1 &\triangleq \texttt{x := x + 2}\\ \mathsf{r}_w &\triangleq \texttt{while (true) \{ skip \}}\\ \mathsf{r} &\triangleq \mathsf{r}_1\texttt{;}\ \mathsf{r}_w\end{aligned}$$

the concrete domain $\mathcal{P}(\mathbb{Z})$, the abstract domains $\mathrm{Int}_{\neq 0}$ (see Example 6.9) and the initial state $P = \{-4, 0\}$. Then $\models_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}\ [\emptyset]$ but this triple cannot be proved in LCL_A extended with (refine-pre).

To show that the triple is (intensionally) valid, we observe that

$$[\![\mathsf{r}]\!]^\sharp_{\mathrm{Int}_{\neq 0}} \alpha(P) = [\![\mathsf{r}_w]\!]^\sharp_{\mathrm{Int}_{\neq 0}} [\![\mathsf{r}_1]\!]^\sharp_{\mathrm{Int}_{\neq 0}} \alpha(P) = \bot$$

because r_w always diverges, so $[\![\mathsf{r}_w]\!]^\sharp_{\mathrm{Int}_{\neq 0}}$ too is the function that always diverge (in the abstract). Therefore,

$$[\![\mathsf{r}]\!]^\sharp_{\mathrm{Int}_{\neq 0}} \alpha(P) = \alpha([\![\mathsf{r}]\!] P) = \alpha(\emptyset) = \bot.$$

To show that the triple is not provable in LCL_A extended with (refine-pre), we rely on two observations.

The first is that all strict subset $P' \subset P$ are such that $\mathrm{Int}_{\neq 0}(P') \subset P$, and the same property holds for all refinements $A' \preceq \mathrm{Int}_{\neq 0}$. To see this, take $P' \subset P$: there are only three such P', and for all of them $\mathrm{Int}_{\neq 0}(P') = P' \subset P$. Moreover, $A' \preceq \mathrm{Int}_{\neq 0}$ means that $A'(P') \subseteq \mathrm{Int}_{\neq 0}(P')$, so

$$A'(P') \subseteq \mathrm{Int}_{\neq 0}(P') = P' \subset P.$$

This property is important because it means that we cannot apply (relax) to change P: to do it, we would need a $P' \subset P$ such that $P \subseteq A'(P')$.

The second is that $[\![\mathsf{r}_1]\!]$ is not locally complete on P in $\mathrm{Int}_{\neq 0}$ or any of its refinements $A' \preceq \mathrm{Int}_{\neq 0}$ such that $A'(P) = \mathrm{Int}_{\neq 0}(P)$:

$$\begin{aligned}A'([\![\mathsf{r}_1]\!]P) &\subseteq \mathrm{Int}_{\neq 0}([\![\mathsf{r}_1]\!]P)\\ &= \{-2, -1, 1, 2\}\\ &\subset \{-2, -1, 0, 1, 2\}\\ &\subseteq A'(\{-2, -1, 0, 1, 2\})\\ &= A'([\![\mathsf{r}_1]\!](\{-4, -3, -2, -1, 0\}))\\ &= A'([\![\mathsf{r}_1]\!](\mathrm{Int}_{\neq 0}(P)))\\ &= A'([\![\mathsf{r}_1]\!]A'(P))\end{aligned}$$

Now suppose to have a derivation of $\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}\ [\emptyset]$. This proof must use (seq) to handle the sequential composition $\mathsf{r}_1\texttt{;}\ \mathsf{r}_w$, so it needs a triple for r_1. By the first observation above, any use of (relax) cannot change the precondition of this triple, even if we resort first to (refine-pre) to refine the domain. Thus we must have a triple $\vdash_{A'} [P]\ \mathsf{r}_1\ [R]$ for some R and $A' \preceq \mathrm{Int}_{\neq 0}$ satisfying $A'(P) = \mathrm{Int}_{\neq 0}(P)$. However, by soundness, any such triple would imply local completeness of $[\![\mathsf{r}_1]\!]$ on P in A', which is a contradiction by the second observation above. ■

Another example of a sound triple which is not provable using (refine-pre), which does not rely on divergence, is in Appendix D, Example D.3. A corollary of these examples

[3] Note that r_w is equivalent to the regular command `false?`.

(and more in general of logical incompleteness) is that there may not exist a refinement A' to carry out the proof using (refine-pre). Another consequence of this incompleteness result is the fact that, even when a command is locally complete in an abstract domain A, we may need to reason about properties that are not expressible in A in order to prove it, as (refine-pre) may not be sufficient.

We now present an example to illustrate that there are situations in which (refine-int) is more practical than (refine-pre), even though they are both able to prove the same triple.

Example 6.11. Consider the two program fragments

$$\mathsf{r}_1 \triangleq \texttt{(y != 0)?; x := y; y := abs(y)}$$
$$\mathsf{r}_2 \triangleq \texttt{x := y; while (x > 1) \{ y := y - 1; x := x - 1 \}}$$

and the program $\mathsf{r} \triangleq \mathsf{r}_1; \mathsf{r}_2$. Consider also the initial state $P = \mathtt{y} \in [-100; 100]$.

This example is a variation of Example 6.9: the difference is the introduction of the relational dependency `x := y` in r_1, that is partially stored in the postcondition R of r_1. Because of this, $\mathrm{Oct}(R)$ and $\mathrm{Int}(R)$ are different, so we cannot apply (refine-pre) to prove $\vdash [R]\ \mathsf{r}_2\ [Q]$ for some Q.

Following Example 6.9, the domain $\mathrm{Int}_{\neq 0}$ is able to infer on r_1 a subset R of the strongest postcondition $\mathtt{y} \in [1; 100] \wedge \mathtt{y} = \mathrm{abs}(\mathtt{x})$ with the same abstraction $\mathrm{Int}_{\neq 0}(R) = [-100; 100]_{\neq 0} \times [1; 100]$. However, for any such R we cannot use (refine-pre) to prove the triple $\vdash_{\mathrm{Int}} [R]\ \mathsf{r}_2\ [\mathtt{x} = 1 \wedge \mathtt{y} = 1]$ via Oct because $\mathrm{Int}(R) = \mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]$ while $\mathrm{Oct}(R) = 1 \leq \mathtt{y} \leq 100 \wedge -\mathtt{y} \leq \mathtt{x} \leq \mathtt{y}$. More in general, any subset of the strongest postcondition contains the relational information $\mathtt{y} = \mathrm{abs}(\mathtt{x})$, so relational domains like octagons and polyhedra [CH78] do not have the same abstraction as the non-relational Int, preventing the use of (refine-pre). However, we can apply (refine-int): considering $R = (\mathtt{y} \in \{1; 2; 100\} \wedge \mathtt{y} = \mathrm{abs}(\mathtt{x}))$, $Q = (\mathtt{x} = 1 \wedge \mathtt{y} = 1)$ and $\mathsf{r}_w \triangleq$ `while (x > 1) { y := y - 1; x := x - 1 }`, we have

$$\begin{aligned}
\mathrm{Int}[\![\mathsf{r}_2]\!]^\sharp_{\mathrm{Oct}}\mathrm{Int}(R) &= \mathrm{Int}[\![\mathsf{r}_2]\!]^\sharp_{\mathrm{Oct}}(\mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]) \\
&= \mathrm{Int}[\![\mathsf{r}_w]\!]^\sharp_{\mathrm{Oct}}[\![\mathtt{x\ :=\ y}]\!]^\sharp_{\mathrm{Oct}}(\mathtt{x} \in [-100; 100] \wedge \mathtt{y} \in [1; 100]) \\
&= \mathrm{Int}[\![\mathsf{r}_w]\!]^\sharp_{\mathrm{Oct}}(1 \leq \mathtt{y} \leq 100, \mathtt{y} = \mathtt{x}) \\
&= \mathrm{Int}(\mathtt{x} = 1 \wedge \mathtt{y} = 1) \\
&= \mathrm{Int}(Q).
\end{aligned}$$

In this example, rule (refine-pre) can be applied to prove the triple, but it requires to have relational information from the assignment `x := y` in r_1, hence forcing the use of a relational domain (e.g. the reduced product [CC79] of Oct and $\mathrm{Int}_{\neq 0}$) for the whole r, making the analysis more expensive. ■

6.3.3 Choosing the refinement

Thanks to the three new rules (refine-ext), (refine-int) and (refine-pre) we can now combine different domains in the same derivation. However, in order to obtain an algorithm that automatises the search of a provable LCL_A triple we are left with the problem of the selection of the right refinement to use each time. A crucial point to the applicability of refine rules is a strategy to find the most convenient refined abstract domain. While we have not addressed this problem yet, we believe there are some interesting starting points in the literature.

In previous sections, we settled the question from a theoretical point of view. Logical completeness results for (refine-ext) (Theorem 6.5) and (refine-int) (Theorem 6.8) implies the existence of a domain in which it is possible to complete the proof (if this were not the case, then the proof could not be completed in any domain, contradicting logical completeness). However, the proofs of those theorems exhibit the concrete domain C as an example, which is unfeasible in general. Dually, as (refine-pre) is logically incomplete (Example 6.10), there are triples that cannot be proved in any domain with it.

As more practical alternatives, we envisage some possibilities. First, we are studying relationships with counterexample-guided abstraction refinement (CEGAR) [Cla+00], which is a technique that exploits refinement in the context of abstract model checking. However, CEGAR and our approach seem complementary. On the one hand, our refinement rules allow a dynamic change of domain, during the analysis and only for a part of it, while CEGAR performs a static refinement and then a new analysis of the whole transition system in the new, more precise domain. On the other hand, our rules lack an instantiation technique, while for CEGAR there are effective algorithms available to pick a suitable refinement.

Second, local completeness shell [BGGR22] were proposed as an analogous of (global) completeness shell [GRS00]. In the article, the authors proposed to use local completeness shells to perform abstract interpretation repair, a technique to refine the abstract domain depending on the program to analyse, just like CEGAR does for abstract model checking. Abstract interpretation repair works well with LCL_A, and could be a way to decide the best refinement for one of our rules in presence of a failed local completeness proof obligation. The advantage of combining repair with our new rules is given by the possibility of discarding the refined domain just after its use in a subderivation instead of using it to carry out the whole derivation. Investigations in this direction is ongoing.

Another related approach, which shares some common ground with CEGAR, is Lazy (Predicate) Abstraction [HJMS02; McM06]. Both ours and this approach exploits different abstract domains for different parts of the proof, refining it as needed. The key difference is that Lazy Abstraction unwinds the control flow graph of the program (with techniques to handle loops) while we work inductively on the syntax. This means that, when Lazy Abstraction refines a domain, it must use it from that point onward (unless it finds a loop invariant). On the other hand, our method can change abstract domain even for different parts of sequential code. However, the technique used in Lazy Abstraction (basically to trace a counterexample back with a theorem prover until it is either found to be spurious or proved to be true) could be applicable to LCL_A: a failed local completeness proof obligation in (transfer) can be traced back with a theorem prover and the failed proof can be used to understand how to refine the abstract domain.

6.4 Locally complete simplification

We now turn our attention to *simplification* of the abstract domain in the LCL_A proof system. It is known that (global) completeness can be achieved both by refining and by simplifying the abstract domain (these construction are called completeness shell and core, respectively [GRS00]). Therefore, we do the same for local completeness. We propose the rule (simplify), in Figure 6.3. This is a dual of (refine-pre): while the latter requires A' to be a refinement of A with the same abstraction on the precondition P, the former requires A' to be a simplification of A with the same abstraction on the postcondition Q. We remark that this rule is independent of the refinement ones: it can be added to LCL_A both with and without any of the refinement rules.

$$\frac{\vdash_{A'} [P]\ \mathsf{r}\ [Q] \quad A' \succeq A \quad A'(Q) = A(Q)}{\vdash_A [P]\ \mathsf{r}\ [Q]}\ (\mathsf{simplify})$$

Figure 6.3: Rule (simplify) for LCL_A.

The proposed rule (simplify) is sound, but differently than the refinement rules, it is so *intensionally*:

Theorem 6.12 (Intensional soundness of rule simplify)**.** *The proof system in Figure 3.2 with the addition of rule* (simplify) *from Figure 6.3 is intensionally sound.*

Proof sketch. Since the proof of Theorem 3.7 in [BGGR21] is by rule induction, we extend its proof with a new inductive case. The full details are in Appendix D. □

This result is somewhat surprising: for completeness, refinement and simplification have the same power; instead, for local completeness, the former appears to be stronger than the latter.

Even though rule (simplify) does not allow the proof system to prove triples which are not intensionally sound, it still allows to prove more triples, as shown in the following example.

Example 6.13. This example builds on the previous Example 6.10. Consider the same program fragments r_1, r_w and r, concrete domain $\mathcal{P}(\mathbb{Z})$, abstract domain $\mathrm{Int}_{\neq 0}$, initial state $P = \{-5, 0\}$ and final state $\emptyset$. We already showed that the triple $\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}\ [\emptyset]$ cannot be proved using LCL_A (extended with (refine-pre)) in Example 6.10. Hence, we only need to show that it is provable using (simplify).

Consider the simplified domain $\mathsf{Div} = \{\bot, \top\} \succeq \mathrm{Int}_{\neq 0}$. The domain Div separates the empty set from any other set, and can be used for divergence analysis. $\llbracket \mathsf{r}_1 \rrbracket$ is locally complete on P in Div:

$$\mathsf{Div}(\llbracket \mathsf{r}_1 \rrbracket(P)) = \top = \mathsf{Div}(\llbracket \mathsf{r}_1 \rrbracket(\top)) = \mathsf{Div}(\llbracket \mathsf{r}_1 \rrbracket(\mathsf{Div}(P)))$$

This means we can prove the triple $\vdash_{\mathsf{Div}} [P]\ \mathsf{r}_1\ [\llbracket \mathsf{r}_1 \rrbracket(P)]$ by just applying (transfer). Moreover, $\llbracket \mathsf{r}_w \rrbracket$ is globally complete since its output is always $\emptyset$. With these two observations, we can derive the triple $\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}\ [\emptyset]$ using (simplify) with the following proof tree:

$$\dfrac{\mathsf{Div} \succeq \mathrm{Int}_{\neq 0} \qquad \dfrac{\dfrac{\mathbb{C}^{\mathsf{Div}}_{P}(\llbracket \mathsf{r}_1 \rrbracket)}{\vdash_{\mathsf{Div}} [P]\ \mathsf{r}_1\ [\llbracket \mathsf{r}_1 \rrbracket(P)]}\ (\mathsf{transfer}) \qquad \dfrac{\mathbb{C}^{\mathsf{Div}}_{\llbracket \mathsf{r}_1 \rrbracket(P)}(\llbracket \mathsf{r}_w \rrbracket)}{\vdash_{\mathsf{Div}} [\llbracket \mathsf{r}_1 \rrbracket(P)]\ \mathsf{r}_w\ [\emptyset]}\ (\mathsf{transfer})}{\vdash_{\mathsf{Div}} [P]\ \mathsf{r}\ [\emptyset]}\ (\mathsf{seq}) \qquad \mathrm{Int}_{\neq 0}(\emptyset) = \emptyset = \mathsf{Div}(\emptyset)}{\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}\ [\emptyset]}\ (\mathsf{simplify})$$

■

Intuitively, in the previous example the incompleteness is caused by the precision of $\mathrm{Int}_{\neq 0}$ on the output of r_1. However, this precision is not needed because the details of the intermediate state are discarded by r_w. The simpler domain Div is able to discard such precision, thus proving local completeness of the composite command.

In this example, we showed that $\vdash_{\mathrm{Int}_{\neq 0}} [P]\ \mathsf{r}\ [\emptyset]$ can be proved in LCL_A extended with (simplify), but by Example 6.10 we know it is not provable by (refine-pre). We remark that the opposite is true as well: LCL_A extended with (refine-pre) can prove extensionally valid triples (cf. Example 6.9) that it cannot prove when extended with (simplify), because the latter is bound by intensional soundness. Together, these two facts means that (refine-pre) and (simplify) extend the logic in two incomparable ways. If we instead include (refine-int), we know by logical completeness (Theorem 6.8) that it can prove all (extensionally) valid triples, including all those provable with (simplify).

However, even though LCL_A extended with (simplify) can prove more triples, it is logically incomplete:

Theorem 6.14 (Intrinsic incompleteness)**.** *Consider the concrete domain of stores $C = \mathcal{P}(\Sigma)$. Assume* Reg *is a Turing complete language, and $A \in \mathrm{Abs}(C)$ is not trivial. Then there exists $P, Q \in C$ and* $\mathsf{r} \in \mathsf{Reg}$ *such that $Q \leq [\![\mathsf{r}]\!] P$, $[\![\mathsf{r}]\!]^\sharp_A \alpha(P) = \alpha(Q)$ but the triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ is not provable in LCL_A extended with* (simplify)*.*

Proof sketch. The proof follows closely that of intrinsic incompleteness of LCL_A [BGGR21, Theorem 5.12] and is reported in Appendix D. □

This theorem shows that the strength of rule (simplify) on the logical level is quite thin. However, this rule could be extremely helpful in practice because it allows to perform part of the analysis in a simpler and possibly much more efficient domain. For instance, consider variable partitioning, a kind of domain simplification. In a series of papers, Singh et al. [SPV15; SPV17b; SPV17a] showed that it leads to great speedups in relational numerical abstract domains, such as octagons and polyhedra. Variable partitioning is a technique that divides variables in subsets such that relations only exists among variables in the same subset. This allows to perform many operations separately on different partitions, reducing the cost that is superlinear in the number of variables (eg. cubic for octagons, exponential for polyhedra). Moreover, partitions are chosen dynamically, so that (1) they change during the analysis and (2) they are guaranteed not to lose precision w.r.t. the non partitioned domain. Let us consider the domain Poly of polyhedra as an example, but we remark these observations are general enough to be applied to many relational numerical domains. Formally, given a set of variables Var and one of its partitions $\pi = \{\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_m\}$, a polyhedron P can be expressed in the partition π if and only if, for all constraints k of P all variables involved in k are in the same element $\mathcal{X}_i \in \pi$. Given a variable partitioning π we denote by Poly_π the abstract domain of polyhedra that can be expressed in the partition π. Clearly $\gamma(\mathsf{Poly}_\pi) \subseteq \gamma(\mathsf{Poly})$, so that $\mathsf{Poly} \preceq \mathsf{Poly}_\pi$ for all π. Since the partition is changed during the analysis, the abstract domain changes too. Rule (refine-ext) cannot accommodate for this change. A domain coarser than all partitions correspond to the maximal partition, in which each variable is on its own, and is non relational: for Poly this domain is Int. In general Int is not precise enough to prove the condition $A[\![\mathsf{r}]\!]^{A'} A(P) = A(Q)$ (instantiated with $A = \mathrm{Int}$ and $A' = \mathsf{Poly}_\pi$ it becomes $\mathrm{Int}[\![\mathsf{r}]\!]^{\mathsf{Poly}_\pi} \mathrm{Int}(P) = \mathrm{Int}(Q)$) at every partition change, since those occur during the analysis and must keep track of relational informations computed up to that point. On the other hand, rule (simplify) perform the "global" analysis in Poly, and simplify locally to Poly_π for the computation, taking advantage of the better performances in the simpler domain. Note that, since the partition is chosen in order not to lose precision w.r.t. the non partitioned domain Poly, the condition $A'(Q) = A(Q)$ is satisfied. This means that variable partitioning can be plugged in LCL_A with our rule (simplify), allowing it to benefit from all the performance increase. While it is intuitive that variable partitioning is sound since it is as precise as Poly, our rule formally justify this claim.

Example 6.15. Consider the program fragments

$$\begin{aligned}
\mathsf{r}_1 &\triangleq \texttt{x := y; y := y - 3; x := x - 4}\\
\mathsf{r}_2 &\triangleq \texttt{z := y; y := 0}\\
\mathsf{r}_3 &\triangleq \texttt{w := z - x}\\
\mathsf{r} &\triangleq \mathsf{r}_1; \mathsf{r}_2; \mathsf{r}_3\\
&= \texttt{x := y; y := y - 3; x := x - 4; z := y; y := 0; w := z - x}
\end{aligned}$$

and the initial state $P = -100 \leq y \leq 100$. At the end, the value for `w` is always 1. To prove it, the analysis must track the dependency between variables using a relational domain such as polyhedra. However, for the first three assignments (i.e., the fragment r_1) we do not need to track any dependency involving `z` and `w`, and after the assignment `y := 0` in r_2 there is no dependency to track with `y`.

Consider the final set of states

$$Q \triangleq (y = 0 \land x \in \{-104, 96\} \land z = x + 1 \land w = 1)$$

and partitions $\pi_1 = \{\{x, y\}, \{z\}, \{w\}\}$ and $\pi_2 = \{\{x, z\}, \{y\}, \{w\}\}$. A proof for the triple $\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}\ [Q]$ can exploit ($\mathsf{simplify}$) to work in Poly_{π_1} and Poly_{π_2} for parts of the program, therefore simplifying the computation of the local completeness proof obligations for ($\mathsf{transfer}$). Fixed $R_1 \triangleq (y \in \{-103, 97\} \land x = y - 1)$ and $R_2 \triangleq (x \in \{-104, 96\} \land y = 0 \land z = x + 1)$, the proof sketch looks like

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash_{\mathsf{Poly}_{\pi_1}} [P]\ \mathsf{r}_1\ [R_1]}}{\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}_1\ [R_1]}\,(\mathsf{simplify}) \quad \dfrac{\cdots}{\vdash_{\mathsf{Poly}} [R_1]\ \mathsf{r}_2\ [R_2]} \quad \dfrac{\dfrac{\cdots}{\vdash_{\mathsf{Poly}_{\pi_2}} [R_2]\ \mathsf{r}_3\ [Q]}}{\vdash_{\mathsf{Poly}} [R_2]\ \mathsf{r}_3\ [Q]}\,(\mathsf{simplify})}{\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}\ [Q]}\,(\mathsf{seq})$$

Note that $\mathsf{Poly}(R_1) = \mathsf{Poly}_{\pi_1}(R_1)$ and $\mathsf{Poly}(Q) = \mathsf{Poly}_{\pi_2}(Q)$ because R_1 (resp. Q) only contains constraints between variables in the same partition of π_1 (resp. π_2). ■

Since ($\mathsf{simplify}$) is analogous to ($\mathsf{refine\text{-}pre}$), a natural question is whether there exists one simplification rule similar to the stronger ($\mathsf{refine\text{-}ext}$). We believe this is not the case. A dual of ($\mathsf{refine\text{-}ext}$) for simplification should involve $[\![\mathsf{r}]\!]^{A'}$, because the goal of the rule is to perform the analysis in A'. The only reasonable input to which we can apply $[\![\mathsf{r}]\!]^{A'}$ is $\alpha'(P)$. Note that, since $A' \succeq A$, abstracting with A before does not change the result: $\alpha' A(P) = \alpha'(P)$. The hypothesis $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ implies, by soundness, $[\![\mathsf{r}]\!]^{A'} \alpha'(P) = \alpha'(Q)$. Moreover, for the rule to be sound its hypotheses must ensure that $\alpha(Q) = \alpha([\![\mathsf{r}]\!] P) = [\![\mathsf{r}]\!]^{A} \alpha(P)$, so any condition involving $[\![\mathsf{r}]\!]^{A'} \alpha'(P)$ and any of those three is equivalent to $A'(Q) = A(Q)$. Based on this argument, we don't expect a weaker condition (that is, a more general rule) for simplification to exist.

6.5 Exploiting convexity

LCL_A requires the existence of a best correct abstraction $\alpha : C \to A$, but there are domains for which only the concretisation map γ is defined (e.g., for polyhedra). If this is the case, LCL_A is not applicable because we cannot even write the (local) completeness equation $Af(P) = AfA(P)$. However, local completeness enjoys abstract convexity (Lemma 3.6). This suggests that, even if a concrete point X cannot be abstracted, we can use a different point P that can be abstracted such that $P \leq X \leq A(P)$. Intuitively, if f is locally complete on P, abstract convexity implies that f is locally complete on x, too. This idea is the basis of the development in this section.

Formally, we introduce an intermediate domain L between C and A such that (1) there is a Galois connection between L and A and (2) we have a monotone concretisation function $\gamma_0 : L \to C$, as in the following diagram:

$$\begin{tikzcd} C & L \arrow[l, "\gamma_0"'] \arrow[r, shift right, "\alpha"'] & A \arrow[l, shift right, "\gamma"'] \end{tikzcd}$$

Intuitively, we take L as the set of all the assertions we can write as pre and postconditions of LCL_A triples, and we limit this to be only a subset of C for which there is an abstraction function $\alpha : L \to A$. For instance, if $A = \mathsf{Poly}$, we can take L to be the set of finite unions of polyhedra, and limit the logic to only use those as pre and postconditions of triples, instead of any possible concrete state. To apply LCL_A using L as the concrete domain, we fix an abstraction $\llbracket \mathsf{r} \rrbracket_L : L \to L$ of $\llbracket \mathsf{r} \rrbracket$. As it is standard in the absence of the abstraction function [CC92], we require $\llbracket \mathsf{r} \rrbracket_L$ to satisfy the soundness condition $\llbracket \mathsf{r} \rrbracket \gamma_0 \leq \gamma_0 \llbracket \mathsf{r} \rrbracket_L$. Since there is a Galois connection between L and A, we can exploit the LCL_A framework (and all of its extensions) using L as the concrete domain and $\llbracket \mathsf{r} \rrbracket_L$ as the concrete semantics. Then , to transfer the nice properties of LCL_A from L to C, we require the concrete point $X \in C$ to be in between the current assertion $P \in L$ and its abstraction $\alpha(P) \in A$. Formally, at any program point with concrete state X and assertion P, we require that

$$\gamma_0(P) \leq X \leq \gamma_0 \gamma \alpha(P) \tag{I}$$

Please note that, if this invariant holds after the execution of a program r, then the Proofs of Verification (Corollary 3.8) holds for the concrete value X.

However, in general this invariant is not preserved by program execution. Given a triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ (w.r.t. the concrete domain L and semantics $\llbracket \mathsf{r} \rrbracket_L$) such that the concrete state before r is X satisfying (I), in general the corresponding invariant after the program $\gamma_0(Q) \leq \llbracket \mathsf{r} \rrbracket X \leq \gamma_0 \gamma \alpha(Q)$ does not hold. This is because $\llbracket \mathsf{r} \rrbracket X$ may not be comparable with $\gamma_0(\llbracket \mathsf{r} \rrbracket_L P)$ even if $\llbracket \mathsf{r} \rrbracket_L$ is sound. To solve this issue, we impose the additional condition of forward completeness [GQ01] of $\llbracket \mathsf{r} \rrbracket_L$, but only on the single point P, namely

$$\gamma_0(\llbracket \mathsf{r} \rrbracket_L P) = \llbracket \mathsf{r} \rrbracket \gamma_0(P).$$

Under this hypothesis, we show the following proposition:

Proposition 6.16. *Assume that $\llbracket \mathsf{r} \rrbracket_L$ is forward complete on P, that $\models_A [P]\ \mathsf{r}\ [Q]$ is valid taking L as the concrete domain and that invariant* (I) *holds. Then*

$$\gamma_0(Q) \leq \llbracket \mathsf{r} \rrbracket X \leq \gamma_0 \gamma \alpha(Q).$$

Proof. We prove the two inequalities separately. We recall that $\llbracket \mathsf{r} \rrbracket$ and γ_0 are monotone, and we use this fact implicitly in the chains of inequalities below.

$$\begin{aligned} \gamma_0(Q) &\leq \gamma_0(\llbracket \mathsf{r} \rrbracket_L P) && [\text{validity of } \vdash_A [P]\ \mathsf{r}\ [Q], \text{ pt. (1)}] \\ &= \llbracket \mathsf{r} \rrbracket \gamma_0(P) && [\text{forward completeness}] \\ &\leq \llbracket \mathsf{r} \rrbracket X && [\gamma_0(P) \leq X] \end{aligned}$$

and

$$\begin{aligned} \llbracket \mathsf{r} \rrbracket X &\leq \llbracket \mathsf{r} \rrbracket \gamma_0 \gamma \alpha(P) && [X \leq \gamma_0 \gamma \alpha(P)] \\ &\leq \gamma_0(\llbracket \mathsf{r} \rrbracket_L \gamma \alpha(P)) && [\text{soundness of } \llbracket \mathsf{r} \rrbracket_L] \\ &\leq \gamma_0 \gamma((\llbracket \mathsf{r} \rrbracket_L)^A \alpha(P)) && [\text{soundness of } (\llbracket \mathsf{r} \rrbracket_L)^A] \\ &= \gamma_0 \gamma \alpha(Q) && [\text{validity of } \vdash_A [P]\ \mathsf{r}\ [Q], \text{ pt. (3)}] \end{aligned}$$

□

This lemma allows us to use LCL_A even in the absence of a best abstraction function. However, to do so we need to identify a set L of assertions which is concrete enough to be forward complete and abstract enough to have an abstraction function $\alpha : L \to A$.

Intuitively, the (local) forward completeness requirement $\gamma_0([\![\mathsf{r}]\!]_L P) = [\![\mathsf{r}]\!]\gamma_0(P)$ implies that all the concrete state traversed are "close enough" to have an abstraction that a proof in LCL_A yields useful information. In other words, the local forward completeness is a "sanity check" that the program did not traverse a concrete state for which the LCL_A theory is not applicable, that are states X for which there exists no $P \in L$ s.t. (I) holds. For instance, with the previous example of polyhedra and finite unions of polyhedra, any shape with a curved "edge" (e.g., a circle) does not satisfy the requirement (I), and the local forward completeness condition ensures such states are not traversed.

Example 6.17. Consider the concrete domain $\mathbb{R}^2$ of two real variables $\texttt{x}$ and $\texttt{y}$, the polyhedra domain Poly as A and the set of finite unions of polyhedra $\mathsf{Poly}_\cup$ as L. Consider the program fragment

$$\mathsf{r}_r \triangleq \texttt{x := (}\sqrt{2}\texttt{ / 2) * x - (}\sqrt{2}\texttt{ / 2) * y; y := x + }\sqrt{2}\texttt{ * y}$$

the program $\mathsf{r} \triangleq \mathsf{r}_r^\star$ and the set of states $X = -5 \leq x \leq 5 \wedge -5 \leq y \leq 5$, describing a square in the Cartesian plane. Please note that the program r_r rotates the point (x, y) in the Cartesian plane by $\pi/4$ radiants, therefore applying r_r to X twice returns X. In particular, letting $X' \triangleq [\![\mathsf{r}_r]\!]X$, this means that $[\![\mathsf{r}]\!]X = X \cup X'$.

Consider the initial assertion $P = \{(5, 5), (-5, 5), (5, -5), (-5, -5)\}$, where a pair denote the value for variables $\texttt{x}$ and $\texttt{y}$, that is the union of four polyhedra, each containing a single point. It holds that $\gamma_0(P) \subseteq X = \gamma_0(\mathsf{Poly}(P))$. We also define $P' = \{(5\sqrt{2}, 0), (-5\sqrt{2}, 0), (0, 5\sqrt{2}), (0, -5\sqrt{2})\}$ and $Q = P \cup P'$. Using $\mathrm{LCL}_{\mathsf{Poly}}$ over $L = \mathsf{Poly}_\cup$, we can first prove the two triples $\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}_r\ [P']$ and $\vdash_{\mathsf{Poly}} [P \cup P']\ \mathsf{r}_r\ [Q]$ using (seq) and ($\mathsf{transfer}$). Then, we compose them using ($\mathsf{iterate}$) and (req) to conclude:

$$\dfrac{\dfrac{\dots}{\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}_r\ [P']} \qquad \dfrac{\dfrac{\dots}{\vdash_{\mathsf{Poly}} [P \cup P']\ \mathsf{r}_r\ [Q]} \quad (P \cup P') \leq \mathsf{Poly}(Q)}{\vdash_{\mathsf{Poly}} [P \cup P']\ \mathsf{r}_r^\star\ [Q]}\ (\mathsf{iterate})}{\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}\ [P \cup Q]}\ (\mathsf{rec})$$

This tells us that the triple $\vdash_{\mathsf{Poly}} [P]\ \mathsf{r}\ [P \cup Q]$ is valid relative to the semantics in $\mathsf{Poly}_\cup$. To transfer the same analysis to the concrete domain $\mathbb{R}^2$, we check forward completeness. In this case, the check is trivial because the semantics of r in L on P, P' and Q is exactly the same as the concrete semantics on $\gamma_0(P)$, $\gamma_0(P')$ and $\gamma_0(Q)$. Note that this triple allows us to identify errors: given the specification $\mathrm{Spec} = x \leq 7$, we observe that $Q \not\leq \mathrm{Spec}$. Since Spec is expressible in Poly, this in turn highlights the error $(5\sqrt{2}, 0) \in Q \setminus \mathrm{Spec}$, which is a true alert because $(5\sqrt{2}, 0) \in X \cup X'$. ■

6.6 Backward analysis

In principle, the theory of Abstract Interpretation does not rely on the analysis being forward. This suggests that LCL can be used for backward analysis, as discussed in [BGGR23, §5.3]. However, they explored the use of **wlp** as the reference backwards semantics, requiring the use of under-approximation abstract domains, that are hard to exploit in practice (see Chapter 4). Instead, our key insight is that by using $[\![\overleftarrow{\cdot}]\!]$ (see equation (5.1)) the proof system can be turned over by duality for backward inference *using classical, over-approximation abstract domains.*

$$\begin{aligned}
\llbracket \overleftarrow{\mathsf{e}} \rrbracket^{\sharp}_{A} a &\triangleq \llbracket \overleftarrow{\mathsf{e}} \rrbracket^{A} a = \alpha \llbracket \overleftarrow{\mathsf{e}} \rrbracket \gamma(a) \\
\llbracket \overleftarrow{\mathsf{r}_1; \mathsf{r}_2} \rrbracket^{\sharp}_{A} a &\triangleq \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket^{\sharp}_{A} \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket^{\sharp}_{A} (a) \\
\llbracket \overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2} \rrbracket^{\sharp}_{A} a &\triangleq \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket^{\sharp}_{A} a \vee A \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket^{\sharp}_{A} a \\
\llbracket \overleftarrow{\mathsf{r}^{\star}} \rrbracket^{\sharp}_{A} a &\triangleq \bigvee_{n \geq 0} (\llbracket \overleftarrow{\mathsf{r}} \rrbracket^{\sharp}_{A})^{n} a
\end{aligned}$$

Figure 6.4: Backward abstract semantics of regular commands.

$$\frac{\mathbb{C}^{A}_{Q}(\llbracket \overleftarrow{\mathsf{c}} \rrbracket)}{\vdash_A \langle\!\langle \llbracket \overleftarrow{\mathsf{c}} \rrbracket Q \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{transfer}) \qquad \frac{P \leq P' \leq A(P) \quad \vdash_A \langle\!\langle P' \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q' \rangle\!\rangle \quad Q' \leq Q \leq A(Q')}{\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{relax})$$

$$\frac{\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle R \rangle\!\rangle \quad \vdash_A \langle\!\langle R \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}{\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}_1; \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{seq}) \qquad \frac{\vdash_A \langle\!\langle P_1 \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle Q \rangle\!\rangle \quad \vdash_A \langle\!\langle P_2 \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}{\vdash_A \langle\!\langle P_1 \vee P_2 \rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{join})$$

$$\frac{\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}^{\star}\ \langle\!\langle R \vee Q \rangle\!\rangle \quad \vdash_A \langle\!\langle R \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}{\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}^{\star}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{rec}) \qquad \frac{P \leq A(Q) \quad \vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}{\vdash_A \langle\!\langle P \vee Q \rangle\!\rangle\ \mathsf{r}^{\star}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{iterate})$$

Figure 6.5: The proof system for CLCL.

We define inductively in Figure 6.4 the backward abstract semantics or regular commands. This definition is analogous to the forward one (Figure 2.5). Thanks to Lemma 5.1 giving $\llbracket \overleftarrow{\cdot} \rrbracket$ the same inductive definition as the forward semantics $\llbracket \cdot \rrbracket$, we can straightforwardly adapt Proposition 2.22 to obtain that $\llbracket \overleftarrow{\cdot} \rrbracket^{\sharp}_{A}$ is a sound abstraction of $\llbracket \overleftarrow{\cdot} \rrbracket$.

We can then define the proof system for Converse Local Completeness Logic (CLCL) in Figure 6.5 by just swapping the roles of pre and post in the LCL rules.

Theorem 6.18 (CLCL is sound). *If the CLCL triple* $\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$ *is provable, then*

1. $P \leq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$,
2. $\alpha(P) = \alpha(\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q)$,
3. $\alpha(P) = \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{\sharp}_{A} \alpha(Q)$.

Just like regular LCL, any provable CLCL triple ensures that $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$ is between P and $A(P)$ and that $\mathbb{C}^{A}_{Q}(\llbracket \overleftarrow{\mathsf{r}} \rrbracket)$. Particularly, this means that if $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \neq \emptyset$ and A is a non-trivial abstraction, then also $P \neq \emptyset$. In other words, given an error Q, the analysis either finds some (non empty) precondition P leading to that Q, or shows that Q is unreachable. On the other end of the spectrum, the abstraction $A(P)$ always exhibits a necessary precondition for Q since $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \subseteq A(\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q) = A(P)$.

Example 6.19. Expanding on Example 5.6, we observe that SIL can also prove the triple $\langle\!\langle R_{2M} \rangle\!\rangle\ \mathsf{r}^{\star}_{w}\ \langle\!\langle R_{2M} \rangle\!\rangle$ via $\langle\!\langle$iter0$\rangle\!\rangle$. However, this would produce the triple $\langle\!\langle$**false**$\rangle\!\rangle$ rloop0 $\langle\!\langle Q_{2M} \rangle\!\rangle$ for the whole program because it finds the post $x = 2000000$ for `x := 0`, that has **false** as the only valid precondition.

To rule out such derivations, we can use CLCL with the octagons domain Oct [Min06]: the triple $\vdash_{\mathrm{Oct}} \langle\!\langle R_{2M} \rangle\!\rangle\ \mathsf{r}^{\star}_{w}\ \langle\!\langle R_{2M} \rangle\!\rangle$ is not valid because $\mathrm{Oct}(R_{2M}) \neq \mathrm{Oct}(\llbracket \overleftarrow{\mathsf{r}^{\star}_{w}} \rrbracket R_{2M})$. This

$$\frac{\vdash_{A'} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle \quad A' \preceq A \quad A(P) = A[\![\overleftarrow{\mathsf{r}}]\!]^{A'} A(Q)}{\vdash_{A} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{refine\text{-}ext})$$

$$\frac{\vdash_{A'} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle \quad A' \succeq A \quad A'(P) = A(P)}{\vdash_{A} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{simplify})$$

$$\frac{\vdash_{A'} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle \quad A' \preceq A \quad A(P) = A[\![\mathsf{r}]\!]^{\sharp}_{A'} A(Q)}{\vdash_{A} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{refine\text{-}int})$$

$$\frac{\vdash_{A'} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle \quad A' \preceq A \quad A'(Q) = A(Q)}{\vdash_{A} \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ (\mathsf{refine\text{-}pre})$$

Figure 6.6: Refinement and simplification rules for CLCL.

way, CLCL forces the analysis to unroll the loop twice, proving the triple

$$\vdash_{\mathrm{Oct}} \langle\!\langle (n > 0 \land x + n \leq 2000000) \lor R_{2M} \rangle\!\rangle\ \mathsf{r}_w^{\star}\ \langle\!\langle R_{2M} \rangle\!\rangle$$

whose full derivation is in Appendix D, Figure D.3. Using this triple, we conclude as in Example 5.6: both applications of $\langle\!\langle$atom$\rangle\!\rangle$ in Figure 5.4 can be replaced by (transfer) since the backward semantics of both assignments is locally complete. Thus, we prove the triple $\vdash_{\mathrm{Oct}} \langle\!\langle \mathbf{true} \rangle\!\rangle$ rloop0 $\langle\!\langle Q_{2M} \rangle\!\rangle$ for the full program, which exposes a manifest error. ■

Other than the original LCL$_A$ proof system, we can adapt all the other extensions proposed in this chapter to CLCL. The fundamental reason that allows this is the fact that $[\![\cdot]\!]$ and $[\![\overleftarrow{\cdot}]\!]$ enjoy the same inductive definition. For instance, (refine-ext) can be adapted with the additional condition $A(P) = A[\![\overleftarrow{\mathsf{r}}]\!]^{A'} A(Q)$. All refinement and simplification rules are presented in Figure 6.6. The refinement rules can be proved *extensionally* sound (whose definition is analogous to LCL$_A$), while the simplification rule is (intensionally) sound.

Proposition 6.20. *The proof system in Figure 6.5 with the addition of any rule from Figure 6.6 is sound.*

Proof sketch. The proof is analogous to those for LCL$_A$. It extends the soundness proof with new inductive cases for the new rules. □

Similarly, if the abstract domain doesn't have an abstraction function α, we can adapt Proposition 6.16 assuming that the invariant holds in for the final state Q and prove it for P, by replacing $[\![\cdot]\!]$ with $[\![\overleftarrow{\cdot}]\!]$ in the proof and using validity of the $\vdash_A \langle\!\langle P \rangle\!\rangle$ r $\langle\!\langle Q \rangle\!\rangle$.

6.7 Summary

In this chapter, we started from LCL$_A$ [BGGR21], a logical framework to prove both correctness and incorrectness of a program combining over and under-approximation in the form of (locally complete) abstractions and Incorrectness Logic. The original work was *intensionally* sound, based on Galois connection and more suitable for forward analysis. We tried to relax those constraints.

For the first, we followed the idea of [BGGR23] to exploit different abstract domains to analyse different portions of the whole program. We propose four new rules that can be

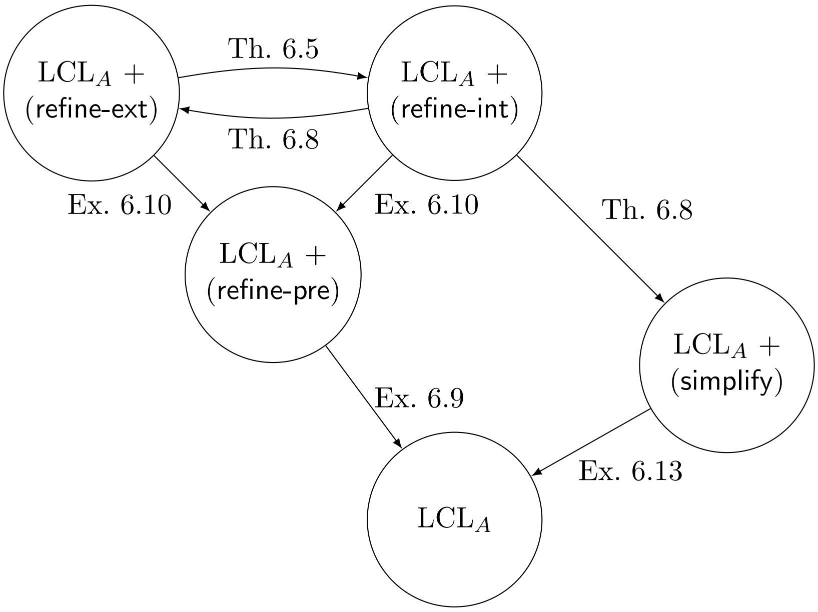


Figure 6.7: Relations between the new proof systems

independently added to the proof system. Three of them are based on domain *refinement*, with different complexity-precision trade-off, and the last is based on domain *simplification*. With any of this rule, we are able to prove many triples that the original LCL$_A$ could not because of how the program is written. For the second, in the absence of α we investigated the possibility to introduce an intermediate domain L between C and A. Under the hypothesis of local forward completeness, we showed that this is enough to recover the properties of LCL$_A$. For the third, we explored the use of the backward SIL together with over-approximation abstract domains instead of IL with under-approximation abstract domains, and we showed that we can recast all results for LCL$_A$ to CLCL.

We present a pictorial comparison among the expressiveness of the various proof systems in Figure 6.7. The bottom node of the diagram represents the original proof system LCL$_A$. Each other node represents the proof system extended with the single rule mentioned in the balloon. Each arrow corresponds to an expressivity result: all triples provable in the target system are also provable in the source system, which is thus more powerful. The labels reference the result that justifies the claim. For simplicity, we omit arrows obtained by transitivity. The two mutual arrows between the two topmost nodes indicate that the two proof systems are logically equivalent (i.e., they can prove the same triples).

Chapter 7

AdjointPDR

In this chapter we study a new PDR-like algorithm (see Section 3.3). Differently than previous approaches, our main tool are *adjunctions*, which we use extensively in our development. We propose a first algorithm, `AdjointPDR`, which exploits an adjoint g to the function f (which roughly identify the backward semantics of f) to quicken the counterexample search. This first algorithm allows us to devise a theory of heuristics to better understand and compare them. However, to apply `AdjointPDR` the right adjoint g to the forward semantics f must exist, and this is not always the case. To get rid of this constraint, we propose `AdjointPDR`$^{\downarrow}$, a variation of `AdjointPDR` which lift the problem to lower sets, where it is always possible to define this adjoint. Lastly, we propose yet another variation of the algorithm, `AdjointPDR`$^{\mathtt{AI}}$, which can instantiate both `AdjointPDR` and `AdjointPDR`$^{\downarrow}$. We implemented this latter algorithm, and compared it against other PDR-like algorithms and state-of-the-art tools with encouraging results.

The content of this chapter is based on [Kor+23].

7.1 Overview

Category theory has recognized adjunctions $f \dashv g$ as fundamental concepts appearing across various mathematical domains [Law69]. Adjointness is prevalent in various branches of computer science as well, including abstract interpretation and functional programming [Lev04]. In our development, we employ adjoints in two distinct ways:

- (Forward-Backward Adjoint) f characterizes the *forward semantics* of a transition system, while g represents the *backward* semantics.
- (Abstraction-Concretization Adjoint) C denotes a concrete semantic domain, while A is an abstract one, akin to abstract interpretation. An adjoint allows us to translate a fixed-point problem from C to A.

The problem we address is the standard lattice-theoretical formulation of safety problems, namely whether the least fixed point of a continuous map b over a complete lattice L is below a given element $p \in L$: $\mu b \leq_? p$.

The first algorithm we present, `AdjointPDR`, assumes the existence of an element $i \in L$ and two adjoints $f \dashv g\colon L \to L$, representing respectively initial states, forward semantics and backward semantics such that $b(x) = f(x) \vee i$ for all $x \in L$.

$$\begin{tikzcd} L \arrow[r, bend right, "g"'] \arrow[r, phantom, "\perp"] & L \arrow[l, bend right, "f"'] \end{tikzcd}$$

Under this assumption, Knaster-Tarski Theorem 2.1 yields the equivalences:

$$\mu b \leq p \quad \Leftrightarrow \quad \mu(f \vee i) \leq p \quad \Leftrightarrow \quad i \leq \nu(g \wedge p),$$

where $\mu(f \vee i)$ and $\nu(g \wedge p)$ are, by Kleene Theorem 2.3, the limits of the *initial* and *final* chains illustrated below.

$$\bot \leq i \leq f(i) \vee i \leq \cdots \qquad\qquad \cdots \leq g(p) \wedge p \leq p \leq \top$$

The distinguishing feature of AdjointPDR is to take as a negative sequence (that is a sequential construction of potential counterexamples) an over-approximation of the final chain. This crucially differs from the negative sequence of other PDR-like algorithm, which is an under-approximation of the computed positive chain.

AdjointPDR is sound (Theorem 7.5) and does not loop (Proposition 7.7), but since the problem $\mu b \leq_? p$ is not always decidable, we cannot prove termination. Nevertheless, AdjointPDR allows for a formal theory of heuristics that are essential when instantiating the algorithm to concrete problems. The theory prescribes the choices to obtain the boundary executions, using initial and final chains (Proposition 7.10); it thus identifies a class of heuristics guaranteeing termination when answers are negative (Theorem 7.15).

In general, AdjointPDR's assumption of a forward-backward adjoint $f \dashv g$ does not hold, especially in probabilistic settings. Our second algorithm AdjointPDR$^{\downarrow}$ circumvents this problem by extending the lattice for the negative sequence, from L to the lattice $L^{\downarrow}$ of *lower sets* in L. Specifically, by using the second form of adjoints, namely an abstraction-concretization pair, the problem $\mu b \leq_? p$ in L can be translated to an equivalent problem on $b^{\downarrow}$ in $L^{\downarrow}$, for which an adjoint $b^{\downarrow} \dashv b^{\downarrow}_r$ always exists.

$$\begin{tikzcd} L \arrow[loop left, "b"] \arrow[r, bend right, "(-)^{\downarrow}"'] \arrow[r, phantom, "\bot"] & L^{\downarrow} \arrow[l, bend right, "\bigsqcup"'] \arrow[loop right, "b^{\downarrow} \dashv b^{\downarrow}_r"] \end{tikzcd}$$

This allows us to run AdjointPDR in the lattice $L^{\downarrow}$. We then notice that the search for a positive chain can be conveniently restricted to principals in $L^{\downarrow}$, which have representatives in L. The resulting algorithm, using L for positive chains and $L^{\downarrow}$ for negative sequences, is AdjointPDR$^{\downarrow}$.

The use of lower sets for the negative sequence is a key advantage. It not only avoids the restrictive assumption of backward adjoint g, but also enables a more thorough search for counterexamples. AdjointPDR$^{\downarrow}$ can simulate stepwise LT-PDR (Theorem 7.23), but it is more general since a single negative sequence in AdjointPDR$^{\downarrow}$ potentially represents multiple (Proposition 7.24) or even all (Proposition 7.25) negative sequences of LT-PDR.

Our lattice-theoretic algorithms yield many concrete instances: the original IC3/PDR as well as Reverse PDR [SS17] are instances of AdjointPDR with L being the powerset of the state space; since LT-PDR can be simulated by AdjointPDR$^{\downarrow}$, the latter generalizes all instances in [Kor+22]. As a notable instance, we apply AdjointPDR$^{\downarrow}$ to MDPs, specifically to decide if the maximum reachability probability [BK08] is below a given threshold. Here the lattice $L = [0,1]^S$ is that of fuzzy predicates over the state space S. Our theory provides guidance to devise two heuristics, for which we prove negative termination (Corollary 7.26).

We implement this latter instance in Haskell. However, the implementation is not based on AdjointPDR$^{\downarrow}$ directly, but rather on a third algorithm, AdjointPDR$^{\mathtt{AI}}$. This can be understood as a generalisation of both AdjointPDR and AdjointPDR$^{\downarrow}$ to a more abstract setting:

$$\begin{tikzcd} (L, \leq_L) \arrow[loop left, "b"] \arrow[r, "\gamma"] & (C, \leq_C) \arrow[loop right, "\bar{b} \dashv \bar{b}_r"] \end{tikzcd}$$

$$\forall j \in [k, n-1],\ x_j \not\leq y_j \qquad \text{(PN)}$$

$$x_0 = \bot \qquad \text{(I0)}$$

$$1 \leq k \leq n \qquad \text{(I1)}$$

$$\forall j \in [0, n-2],\ x_j \leq x_{j+1} \qquad \text{(I2)}$$

$$\forall j \in [0, n-1],\ (f \vee i)^j(\bot) \leq x_j \leq (g \wedge p)^{n-1-j}(\top) \qquad \text{(A1)}$$

$$\forall j \in [1, n-1],\ x_{j-1} \leq g^{n-1-j}(p) \qquad \text{(A2)}$$

$$\forall j \in [k, n-1],\ g^{n-1-j}(p) \leq y_j \qquad \text{(A3)}$$

$$i \leq x_1 \qquad \text{(P1)}$$

$$x_{n-2} \leq p \qquad \text{(P2)}$$

$$\forall j \in [0, n-2],\ f(x_j) \leq x_{j+1} \qquad \text{(P3)}$$

$$\forall j \in [0, n-2],\ x_j \leq g(x_{j+1}) \qquad \text{(P3a)}$$

$$\text{If } \vec{y} \neq \varepsilon \text{ then } p \leq y_{n-1} \qquad \text{(N1)}$$

$$\forall j \in [k, n-2],\ g(y_{j+1}) \leq y_j \qquad \text{(N2)}$$

Figure 7.1: Invariants of `AdjointPDR`.

where $\gamma\colon L \to C$ is an order embedding and $b, \overline{b}$ and γ are required to satisfy a condition that is known in the setting of abstract interpretation as *forward completeness* [GRS00].

We experimentally evaluate our implementation. We compare it against existing probabilistic PDR algorithms (PrIC3 [Bat+20], LT-PDR [Kor+22]) and a non-PDR one (Storm [DJKV17]). The performance of `AdjointPDR`$^{\downarrow}$ is encouraging—it supports the potential of PDR algorithms in probabilistic model checking. The experiments also indicate the importance of having a variety of heuristics, and thus the value of our adjoint framework that helps in coming up with those. Additionally, we found that abstraction features of Haskell allow us to code lattice-theoretic algorithms almost literally (∼100 lines). Implementing a few heuristics takes another ∼240 lines. This way, we found that mathematical abstraction can directly help in easing implementation effort.

7.2 Adjoint PDR

In this section we introduce `AdjointPDR`, an algorithm that takes in input a tuple (i, f, g, p) with $i, p \in L$ and $f \dashv g\colon L \to L$ and, if it terminates, it returns true whenever $\mathrm{lfp}(f \vee i) \leq p$ and false otherwise. The algorithm manipulates two sequences of elements of L:

$$\vec{x} \triangleq x_0, \ldots, x_{n-1} \qquad \vec{y} \triangleq y_k, \ldots y_{n-1}$$

of length n and $n-k$, respectively. These satisfy, through the executions of `AdjointPDR`, the invariants in Figure 7.1. By (A1), x_j over-approximates the j-th element of the initial chain, namely $(f \vee i)^j(\bot) \leq x_j$, while, by (A3), the j-indexed element y_j of $\vec{y}$ over-approximates $g^{n-j-1}(p)$ that, borrowing the terminology of Example 3.11, is the set of states which are safe in $n-j-1$ transitions. Moreover, by (PN), the element y_j witnesses that x_j is unsafe, i.e., that $x_j \not\leq g^{n-1-j}(p)$ or equivalently $f^{n-j-1}(x_j) \not\leq p$. Notably, $\vec{x}$ is a positive chain and $\vec{y}$ a negative sequence, according to the definitions below.

Definition 7.1 (positive chain)**.** A *positive chain* for $\mathrm{lfp}(f \vee i) \leq p$ is a finite chain $x_0 \leq \cdots \leq x_{n-1}$ in L of length $n \geq 2$ which satisfies (P1), (P2), (P3) in Figure 7.1. It is *conclusive* if $x_{j+1} \leq x_j$ for some $j \leq n-2$.

In a conclusive positive chain, x_{j+1} provides an invariant for $f \vee i$ and thus, by (2.2), $\mathrm{lfp}(f \vee i) \leq p$ holds. So, when $\vec{x}$ is conclusive, `AdjointPDR` returns true.

Definition 7.2 (negative sequence)**.** A *negative sequence* for $\mathrm{lfp}(f \vee i) \leq p$ is a finite sequence $y_k, \ldots, y_{n-1}$ in L with $1 \leq k \leq n$ which satisfies (N1) and (N2) in Figure 7.1. It is *conclusive* if $k = 1$ and $i \not\leq y_1$.

```
                              AdjointPDR $(i, f, g, p)$
<INITIALISATION>
  $(\vec{x}\|\vec{y})_{n,k}$ := $(\bot, \top\|\varepsilon)_{2,2}$
<ITERATION>                                          % $\vec{x}, \vec{y}$ not conclusive
  case $(\vec{x}\|\vec{y})_{n,k}$ of
      $\vec{y} = \varepsilon$ and $x_{n-1} \leq p$ :                %(Unfold)
          $(\vec{x}\|\vec{y})_{n,k}$ := $(\vec{x}, \top\|\varepsilon)_{n+1,n+1}$
      $\vec{y} = \varepsilon$ and $x_{n-1} \not\leq p$ :                %(Candidate)
          choose $z \in L$ such that $x_{n-1} \not\leq z$ and $p \leq z$;
          $(\vec{x}\|\vec{y})_{n,k}$ := $(\vec{x}\|z)_{n,n-1}$
      $\vec{y} \neq \varepsilon$ and $f(x_{k-1}) \not\leq y_k$ :            %(Decide)
          choose $z \in L$ such that $x_{k-1} \not\leq z$ and $g(y_k) \leq z$;
          $(\vec{x}\|\vec{y})_{n,k}$ := $(\vec{x}\|z, \vec{y})_{n,k-1}$
      $\vec{y} \neq \varepsilon$ and $f(x_{k-1}) \leq y_k$ :            %(Conflict)
          choose $z \in L$ such that $z \leq y_k$ and $(f \vee i)(x_{k-1} \wedge z) \leq z$;
          $(\vec{x}\|\vec{y})_{n,k}$ := $(\vec{x} \wedge_k z\|\mathsf{tail}(\vec{y}))_{n,k+1}$
  endcase
<TERMINATION>
    if $\exists j \in [0, n-2] \,.\, x_{j+1} \leq x_j$ then return true    % $\vec{x}$ conclusive
    if $i \not\leq y_1$ then return false                         % $\vec{y}$ conclusive
```

Figure 7.2: `AdjointPDR` algorithm checking $\mathrm{lfp}(f \vee i) \leq p$.

When $\vec{y}$ is conclusive, `AdjointPDR` returns false as y_1 provides a counterexample: (N1) and (N2) entail (A3) and thus $i \not\leq y_1 \geq g^{n-2}(p)$, so that $g^{n-2}(p) \geq \mathrm{gfp}(g \wedge p)$ and thus $i \not\leq \mathrm{gfp}(g \wedge p)$. By (2.1), $\mathrm{lfp}(f \vee i) \not\leq p$.

The pseudocode of the algorithm is in Figure 7.2, where we write $(\vec{x}\|\vec{y})_{n,k}$ to compactly represents the state of the algorithm: the pair (n, k) is called the *index* of the state, with $\vec{x}$ of length n and $\vec{y}$ of length $n-k$. When $k = n$, $\vec{y}$ is the empty sequence ε. For any $z \in L$, we write $\vec{x}, z$ for the chain $x_0, \ldots, x_{n-1}, z$ of length $n+1$ and $z, \vec{y}$ for the sequence $z, y_k, \ldots y_{n-1}$ of length $n-(k-1)$. Moreover, we write $\vec{x} \wedge_j z$ for the chain $x_0 \wedge z, \ldots, x_j \wedge z, x_{j+1}, \ldots, x_{n-1}$. Finally, $\mathsf{tail}(\vec{y})$ stands for the tail of $\vec{y}$, namely $y_{k+1}, \ldots y_{n-1}$ of length $n - (k+1)$.

The algorithm starts in the initial state $s_0 \triangleq (\bot, \top\|\varepsilon)_{2,2}$ and, unless one of $\vec{x}$ and $\vec{y}$ is conclusive, iteratively applies one of the four mutually exclusive rules: (Unfold), (Candidate), (Decide) and (Conflict). The rule (Unfold) extends the positive chain by one element when the negative sequence is empty and the positive chain is under p; since the element introduced by (Unfold) is $\top$, its application typically triggers rule (Candidate) that starts the negative sequence with an over-approximation of p. Recall that the role of y_j is to witness that x_j is unsafe. After (Candidate) either (Decide) or (Conflict) are possible: if y_k witnesses that, besides x_k, also $f(x_{k-1})$ is unsafe, then (Decide) is used to further extend the negative sequence to witness that x_{k-1} is unsafe; otherwise, the rule (Conflict) improves the precision of the positive chain in such a way that y_k no longer witnesses $x_k \wedge z$ unsafe and, thus, the negative sequence is shortened. Note that, in (Candidate), (Decide) and (Conflict), the element $z \in L$ is chosen among a set of possibilities, thus `AdjointPDR` is nondeterministic.

To illustrate the executions of the algorithm, we adopt a labeled transition system notation. Let $\mathcal{S} \triangleq \{(\vec{x}\|\vec{y})_{n,k} \mid n \geq 2,\ k \leq n,\ \vec{x} \in L^n \text{ and } \vec{y} \in L^{n-k}\}$ be the set of all possible states of `AdjointPDR`. We call $(\vec{x}\|\vec{y})_{n,k} \in \mathcal{S}$ *conclusive* if $\vec{x}$ or $\vec{y}$ are such. When $s \in \mathcal{S}$ is not conclusive, we write $s \xrightarrow{D}$ to mean that s satisfies the guards in the rule

(Decide), and $s \overset{D}{\to}_z s'$ to mean that, being (Decide) applicable, `AdjointPDR` moves from state s to s' by choosing z. Similarly for the other rules: the labels Ca, Co and U stands for (Candidate), (Conflict) and (Unfold), respectively. When irrelevant we omit to specify labels and choices and we just write $s \to s'$. As usual $\to^+$ stands for the transitive closure of $\to$ and $\to^*$ stands for the reflexive and transitive closure of $\to$.

Example 7.3. Consider the safety problem in Example 3.11. Below we illustrate two possible computations of `AdjointPDR` that differ for the choice of z in (Conflict). The first run is conveniently represented as the following series of transitions.

$$\begin{aligned}
&(\emptyset, S \| \varepsilon)_{2,2} \overset{Ca}{\to}_P (\emptyset, S \| P)_{2,1} \overset{Co}{\to}_I (\emptyset, I \| \varepsilon)_{2,2}\\
&\overset{U}{\to} (\emptyset, I, S \| \varepsilon)_{3,3} \overset{Ca}{\to}_P (\emptyset, I, S \| P)_{3,2} \overset{Co}{\to}_{S_2} (\emptyset, I, S_2 \| \varepsilon)_{3,3}\\
&\overset{U}{\to}\overset{Ca}{\to}_P (\emptyset, I, S_2, S \| P)_{4,3} \overset{Co}{\to}_{S_3} (\emptyset, I, S_2, S_3 \| \varepsilon)_{4,4}\\
&\overset{U}{\to}\overset{Ca}{\to}_P (\emptyset, I, S_2, S_3, S \| P)_{5,4} \overset{Co}{\to}_{S_4} (\emptyset, I, S_2, S_3, S_4 \| \varepsilon)_{5,5}\\
&\overset{U}{\to}\overset{Ca}{\to}_P (\emptyset, I, S_2, S_3, S_4, S \| P)_{6,5} \overset{Co}{\to}_{S_4} (\emptyset, I, S_2, S_3, S_4, S_4 \| \varepsilon)_{6,6}
\end{aligned}$$

The last state returns true since $x_4 = x_5 = S_4$. Observe that the chain $\vec{x}$, with the exception of its last element x_{n-1}, is exactly the initial chain of $(T \cup I)$, i.e., x_j is the set of states reachable in at most $j-1$ steps. In the second computation, the elements of $\vec{x}$ are roughly those of the final chain of $(G \cap P)$. More precisely, after (Unfold) or (Candidate), x_{n-j} for $j < n-1$ is the set of states which only reach safe states within j steps.

$$\begin{aligned}
&(\emptyset, S \| \varepsilon)_{2,2} \overset{Ca}{\to}_P (\emptyset, S \| P)_{2,1} \overset{Co}{\to}_P (\emptyset, P \| \varepsilon)_{2,2}\\
&\overset{U}{\to}\overset{Ca}{\to}_P (\emptyset, P, S \| P)_{3,2} \overset{D}{\to}_{S_4} (\emptyset, P, S \| S_4, P)_{3,1} \overset{Co}{\to}_{S_4} (\emptyset, S_4, S \| P)_{3,2} \overset{Co}{\to}_P (\emptyset, S_4, P \| \varepsilon)_{3,3}\\
&\overset{U}{\to}\overset{Ca}{\to}_P (\emptyset, S_4, P, S \| P)_{4,3} \overset{D}{\to}_{S_4} (\emptyset, S_4, P, S \| S_4, P)_{4,2} \overset{Co}{\to}_{S_4} (\emptyset, S_4, S_4, S \| P)_{4,3}
\end{aligned}$$

Observe that, by invariant (A1), the values of $\vec{x}$ in the two runs are, respectively, the least and the greatest values for all possible computations of `AdjointPDR`. ■

7.3 Properties of `AdjointPDR`

In this section we prove the main properties of `AdjointPDR`: (1) any returned result is valid (soundness); (2) although `AdjointPDR` can diverge, any state is never visited twice (called progression); (3) certain heuristics can be used to guarantee termination when a counterexample exists (called negative termination).

7.3.1 Invariants

The proofs of the properties of `AdjointPDR` rely on the properties in Figure 7.1. In this section, we prove that such properties are invariants:

Proposition 7.4. *For any possible choice performed by* `AdjointPDR`*, the properties in Figure 7.1 hold in all reachable states of the algorithm.*

In proving the invariants, some observations on the choice of element z naturally emerge. First, the proofs of the three invariants (I0), (I1) and (I2) do not rely on the properties of the chosen element $z \in L$. For proving the invariants of the positive chain ((P1), (P2), (P3) and (P3a)) and of the negative sequence ((N1) and (N2)) we only exploit the *second* constraints on z of each rule of the algorithm, namely $p \leq z$ in (Candidate), $g(y_k) \leq z$ in (Decide), and $(f \vee i)(x_{k-1} \wedge z) \leq z$ in (Conflict). Lastly, the *first* constraint on z in each rule ensures the remaining invariants ((PN), (A1), (A2) and (A3)), which in turn are key to the proof of progression.

To make the proofs more uniform and compact, we adopt the following notation: for a state s and a property (Q) we will write $s \models (Q)$ to mean that (Q) holds in s. We will often show that (Q) is an invariant inductively: namely, we will prove

(a) $s_0 \models (Q)$ and

(b) if $s \models (Q)$ and $s \to s'$, then $s' \models (Q)$.

Hereafter, we fix $s = (\vec{x} \| \vec{y})_{n,k}$ and $s' = (\vec{x}' \| \vec{y}')_{n',k'}$. As usual we will write x_j and y_j for the elements of $\vec{x}$ and $\vec{y}$. For the elements of $\vec{x}'$ and $\vec{y}'$, we will write x'_j and y'_j. Throughout the proofs, we will avoid to repeat every time in (b) that $s \models (Q)$, and we will just write $\overset{(Q)}{=}$ or $\overset{(Q)}{\leq}$ whenever using such hypothesis. Moreover in (b) we will avoid to specify those cases that are trivial: for instance, for the properties that only concerns the positive chain $\vec{x}$, e.g., (I0) and (P3), it is enough to check the property (b) for $s \overset{U}{\to} s'$ and $s \overset{Co}{\to} s'$, since $s \overset{D}{\to} s'$ and $s \overset{Ca}{\to} s'$ only modify the negative sequence $\vec{y}$. We illustrate below only the most interesting cases. The remaining ones are in Appendix E.

Proof sketch. **Case** (I0): $x_0 = \bot$

(a) In s_0, $x_0 = \bot$.

(b) If $s \overset{U}{\to} s'$, then $x'_0 = x_0 \overset{\text{(I0)}}{=} \bot$.
If $s \overset{Co}{\to}_z s'$, then $x'_0 = x_0 \wedge z \overset{\text{(I0)}}{=} \bot \wedge z = \bot$.

Case (I1): $1 \leq k \leq n$

- To prove that $1 \leq k$, observe that k is initialised at 2 and that it is only decremented by 1. When $k = 1$, $\vec{y} \neq \varepsilon$. By (I0) $x_0 = \bot$. Since f is a left adjoint, $f(\bot) = \bot$. Thus, $f(x_0) \leq y_1$. This means that either the state is conclusive and the algorithm returns, or (Conflict) is enabled and thus k is incremented.

- To prove that $k \leq n$, observe that k is incremented only by 1. When $k = n$, the algorithm does either (Unfold) or (Candidate). In the latter case, k is decremented. In the former, both n and k are incremented.

Case (P3): $\forall j \in [0, n-2]$, $f(x_j) \leq x_{j+1}$

(a) In s_0, since $n = 2$ one needs to check only the case $j = 0$: $f(x_0) \leq \top = x_1$.

(b) If $s \overset{U}{\to} s'$, then $f(x'_j) = f(x_j) \overset{\text{(I2)}}{\leq} x_{j+1} = x'_{j+1}$ for all $j \in [0, n-2]$. For $j = n-1$, $f(x'_{n-1}) = f(x_{n-1}) \leq \top = x'_{j+1}$. Since $n' = n+1$, then $\forall j \in [0, n'-2]$, $f(x'_j) \leq x'_{j+1}$.
If $s \overset{Co}{\to}_z s'$, since $f(x_{k-1} \wedge z) \leq z$, then by (I2) and monotonicity of f it holds that $\forall j \in [0, k-1]$, $f(x_j \wedge z) \leq z$. Since $f(x_j \wedge z) \leq f(x_j) \overset{\text{(P3)}}{\leq} x_{j+1}$, it holds that $f(x_j \wedge z) \leq x_{j+1} \wedge z$ for all $j \in [0, k-1]$. With this observation is immediate to conclude that $\forall j \in [0, n'-2]$, $f(x'_j) \leq x'_{j+1}$.

Case (N2): $\forall j \in [k, n-2]$, $g(y_{j+1}) \leq y_j$
The case of (Conflict) is trivial: the negative sequence $\vec{y}$ is truncated in the rule (Conflict), and if the invariant holds for $\vec{y}$ then it holds for its tail $\mathsf{tail}(\vec{y})$ as well.

(a) In s_0, $k = 2$ and $n = 2$. Thus (N2) trivially holds.

(b) If $s \overset{Ca}{\to} s'$, then $k' = n - 1$ and thus (N2) trivially holds.
If $s \overset{D}{\to}_z s'$, since $z \geq g(y_k)$ and $k' = k - 1$, then $y'_{k'} = y'_{k-1} = z \geq g(y_k) = g(y'_k) = g(y'_{k'+1})$. For $j \in [k'+1, n-2]$, namely for $j \in [k, n-2]$, it holds that $y'_j = y_j \overset{(N2)}{\geq} g(y_{j+1}) = g(y'_{j+1})$. Thus, $\forall j \in [k', n-2]$, $g(y'_{j+1}) \leq y'_j$.

Case (PN): $\forall j \in [k, n-1]$, $x_j \not\leq y_j$

(a) In s_0, $k = n$ and thus (PN) trivially holds.

(b) If $s \overset{U}{\to} s'$, then $k' = n'$ and thus (PN) trivially holds.
If $s \overset{Ca}{\to}_z s'$, since $x_{n-1} \not\leq z$, $x'_{n-1} = x_{n-1}$ and $k' = n' - 1 = n - 1$, then $x'_{n'-1} = x_{n-1} \not\leq z = y'_{n'-1}$.
If $s \overset{D}{\to}_z s'$, since $x_{k-1} \not\leq z$, then $x'_{k-1} = x_{k-1} \not\leq z = y'_{k-1}$. Moreover, $\forall j \in [k, n-1]$, $x'_j = x_j \overset{(PN)}{\not\leq} y_j = y'_j$. Thus, $\forall j \in [k', n'-1]$, $x'_j \not\leq y'_j$.
If $s \overset{Co}{\to} s'$, then $k' = k + 1$ and $n' = n$. Observe that for $j \in [k+1, n-1]$, $x'_j = x_j \overset{(PN)}{\not\leq} y_j = y'_j$. Thus $\forall j \in [k', n'-1]$, $x'_j \not\leq y'_j$.

□

7.3.2 Soundness

Once the properties in Figure 7.1 are proved to be invariants, the proof of soundness of AdjointPDR is rather straightforward: it only appeals to the Knaster-Tarski fixed-point theorem for the positive case, and to the Kleene one for the negative case.

Theorem 7.5 (Soundness). AdjointPDR *is sound, namely,*

1. *If* AdjointPDR *returns true then lfp*$(f \vee i) \leq p$.
2. *If* AdjointPDR *returns false then lfp*$(f \vee i) \not\leq p$.

Proof. We prove the two items separately.

1. Observe that AdjointPDR returns true if $x_{j+1} \leq x_j$. By (P3), we thus have $f(x_j) \leq x_{j+1} \leq x_j$. Moreover, by (P1) and (I2), it holds that $i \leq x_j$ and $x_j \leq p$. Therefore, it holds that
$$(f \vee i)x_j \leq x_j \leq p.$$
By (2.2), we have that $\mathrm{lfp}(f \vee i) \leq p$.

2. Observe that AdjointPDR returns false if $i \not\leq y_1$. By (A3), $g^{n-2}(p) \leq y_1$. Thus $i \not\leq g^{n-2}(p)$. Moreover
$$\begin{aligned} g^{n-2}p &\leq \bigwedge_{j \in \omega} g^j(p) \\ &= \mathrm{gfp}(g \wedge p) \end{aligned}$$
Thus $i \not\leq \mathrm{gfp}(g \wedge p)$. By (2.1), $\mathrm{lfp}(f \vee i) \not\leq p$.

□

7.3.3 Progression

It is necessary to prove that in any step of the execution, if the algorithm does not return true or false, then it can progress to a new state, not yet visited. To this aim we must deal with the subtleties of the non-deterministic choice of the element z in (Candidate), (Decide) and (Conflict). The following proposition ensures that, for any of these three rules, there is always a possible choice.

Proposition 7.6 (Canonical choices)**.** *The following choices of z are always possible:*

1. *in (Candidate) $z = p$;*
2. *in (Decide) $z = g(y_k)$;*
3. *in (Conflict) $z = y_k$;*
4. *in (Conflict) $z = (f \vee i)(x_{k-1})$.*

Thus, for all non-conclusive $s \in \mathcal{S}$, if $s_0 \to^* s$ then $s \to$.

Proof. For each rule, we prove that if the guard of the rule is satisfied then the choice of z satisfies the required constraints.

1. The guard of (Candidate) is $x_{n-1} \not\leq p$. By choosing $z = p$, one has that $x_{n-1} \not\leq z$ and $p \leq z$ are trivially satisfied;
2. The guard of (Decide) is $f(x_{k-1}) \not\leq y_k$ thus, by $f \dashv g$, $x_{k-1} \not\leq g(y_k)$. By choosing $z = g(y_k)$, one has that $x_{k-1} \not\leq z$ and $g(y_k) \leq z$;

The proofs for the choices in (Conflict) are more subtle. First of all, observe that if $k = 1$, then $i \leq y_1$ otherwise the algorithm would have returned false. Moreover, for $k \geq 2$, we have that $i \leq x_{k-1} \leq y_k$: the first inequality holds by (P1) and the second by (A2) and (A3). In summary,

$$\forall j \geq 1 \, . \, i \leq y_j. \tag{7.1}$$

We can then proceed as follows.

3. The guard of (Conflict) is $f(x_{k-1}) \leq y_k$. By choosing $z = y_k$, one has that $z \leq y_k$ trivially holds. For $(f \vee i)(x_{k-1} \wedge z) \leq z$ observe that

$$\begin{aligned}(f \vee i)(x_{k-1} \wedge z) &= f(x_{k-1} \wedge z) \vee i && \text{[def.]}\\ &\leq f(x_{k-1}) \vee i && \text{[monotonicity]}\\ &\leq z \vee i && \text{[guard]}\\ &= z && \text{[(7.1)]}\end{aligned}$$

4. The guard of (Conflict) is $f(x_{k-1}) \leq y_k$. By choosing $z = (f \vee i)(x_{k-1})$, one has that $(f \vee i)(x_{k-1} \wedge z) \leq (f \vee i)(x_{k-1}) = z$ holds by monotonicity. For $z \leq y_k$, by using the guard and (7.1), we have that $z = (f \vee i)(x_{k-1}) = f(x_{k-1}) \vee i \leq y_k$.

□

The following proposition ensures that `AdjointPDR` always traverses new states.

Proposition 7.7 (Impossibility of loops)**.** *If $s_0 \to^* s \to^+ s'$, then $s \neq s'$.*

Proof. Let us consider the following partial order on positive chains: given two sequences $\vec{x} = x_0, \dots x_{n-1}$ and $\vec{x}' = x'_0, \dots, x'_{n'-1}$, we say $\vec{x} \preceq \vec{x}'$ if

$$n \leq n' \land x_j \geq x'_j \text{ for each } j \in [0, n-1]$$

We extend the order to states by letting $(\vec{x} \| \vec{y})_{n,k} \preceq (\vec{x}' \| \vec{y}')_{n',k'}$ with $\vec{x} \prec \vec{x}'$ or $\vec{x} = \vec{x}'$ and $k \geq k'$.

We prove the statement by showing that applying a rule strictly increases the state in that partial order. As before, we use non-primed variables such as $\vec{x}$ for values before the application of a rule, and primed variables such as $\vec{x}'$ after.

For (Unfold), we have that $n < n' = n + 1$ and $x_j = x'_j$ for each $j \in [0, n-1]$.

For (Candidate), we have $\vec{x}' = \vec{x}$ and $k' = n - 1 < n = k$.

For (Decide), we have $\vec{x}' = \vec{x}$ and $k' = k - 1 < k$.

For (Conflict), $n = n'$, and

$$x'_j = \begin{cases} x_j & \text{if } j > k \\ x_j \land z & \text{if } j \leq k \end{cases}$$

So for $j \in [k+1, n-1]$ we have $x_j = x'_j$, and for $j \in [0, k]$ we have $x_j \geq x_j \land z = x'_j$. So $\vec{x} \preceq \vec{x}'$. Assume by contradiction that $x'_k = x_k$. Since $x'_k = x_k \land z$, this is equivalent to $x_k \leq z$. The choice of z in (Conflict) satisfies $z \leq y_k$, that would imply $x_k \leq z \leq y_k$. However, this is a contradiction, since by (PN) we know $x_k \not\leq y_k$. Hence $x_k \geq x'_k$, meaning $\vec{x} \prec \vec{x}'$. □

Observe that the above propositions entail that AdjointPDR terminates whenever the lattice L is finite, since the set of reachable states is finite in this case.

Example 7.8. For (I, T, G, P) as in Example 3.11, AdjointPDR behaves essentially as IC3/PDR [Bra11], solving reachability problems for transition systems with finite state space S. Since the lattice $\mathcal{P}S$ is also finite, AdjointPDR always terminates. ■

7.3.4 Heuristics

The nondeterministic choices of the algorithm can be resolved by using heuristics. Intuitively, a heuristic chooses for any states $s \in \mathcal{S}$ an element $z \in L$ to be possibly used in (Candidate), (Decide) or (Conflict), so it is just a function $h \colon \mathcal{S} \to L$. When defining a heuristic, we will avoid to specify its values on conclusive states or in those performing (Unfold), as they are clearly irrelevant.

With a heuristic, one can instantiate AdjointPDR by making the choice of z as prescribed by h. Syntactically, this means to erase from the code of Figure 7.2 the three lines of choose and replace them with z:= $h(\,(\vec{x} \| \vec{c})_{n,k}\,)$. We call AdjointPDR$_h$ the resulting deterministic algorithm and write $s \to_h s'$ to mean that AdjointPDR$_h$ moves from state s to s'. We let $\mathcal{S}^h \triangleq \{s \in \mathcal{S} \mid s_0 \to_h^* s\}$ be the sets of all states reachable by AdjointPDR$_h$.

Definition 7.9 (legit heuristic)**.** A heuristic $h \colon \mathcal{S} \to L$ is called *legit* whenever for all $s, s' \in \mathcal{S}^h$, if $s \to_h s'$ then $s \to s'$.

When h is legit, the only execution of the deterministic algorithm AdjointPDR$_h$ is one of the possible executions of the non-deterministic algorithm AdjointPDR.

The canonical choices provide two legit heuristics: first, we call *simple* any legit heuristic h that chooses z in (Candidate) and (Decide) as in Proposition 7.6:

$$(\vec{x} \| \vec{y})_{n,k} \mapsto \begin{cases} p & \text{if } (\vec{x} \| \vec{y})_{n,k} \overset{Ca}{\to} \\ g(y_k) & \text{if } (\vec{x} \| \vec{y})_{n,k} \overset{D}{\to} \end{cases} \tag{7.2}$$

Then, if the choice in (Conflict) is like in Proposition 7.6.4, we call h *initial*; if it is like in Proposition 7.6.3, we call h *final*. Shortly, the two legit heuristics are:

simple initial	(7.2) and $(\vec{x}\|\vec{y})_{n,k} \mapsto (f \vee i)(x_{k-1})$	if $(\vec{x}\|\vec{y})_{n,k} \in Co$
simple final	(7.2) and $(\vec{x}\|\vec{y})_{n,k} \mapsto y_k$	if $(\vec{x}\|\vec{y})_{n,k} \in Co$

Interestingly, with any simple heuristic, the sequence $\vec{y}$ takes a familiar shape:

Proposition 7.10. *Let $h\colon \mathcal{S} \to L$ be any simple heuristic. For all $(\vec{x}\|\vec{y})_{n,k} \in \mathcal{S}^h$, invariant* (A3) *holds as an equality, namely for all $j \in [k, n-1]$, $y_j = g^{n-1-j}(p)$.*

Proof. As for the invariants, we prove this equality by induction showing

(a) it holds for s_0 and

(b) if it holds for s and $s \to s'$, then it holds for s'.

In s_0 and after (Unfold), since $k = n$ there is no $j \in [k, n-1]$.

For (Conflict), since the property holds on $\vec{y}$ it also holds on $\vec{y}' = \mathsf{tail}(\vec{y})$.

For (Candidate), $\vec{y}' = p$ and $k' = n-1$, so the thesis holds because $y_{n-1} = p = g^{n-1-(n-1)}p$.

For (Decide), $\vec{y}' = g(y_k), \vec{y}$ and $k' = k-1$. For all $j \in [k'+1, n-1]$ the thesis holds because $y'_j = y_j$. For $j = k'$, we have $y_{k'} = g(y_k) = g(g^{n-1-k}(p)) = g^{n-1-k'}(p)$. □

By the above proposition and (A3), the negative sequence $\vec{y}$ occurring in the execution of `AdjointPDR`$_h$, for a simple heuristic h, is the least amongst all the negative sequences occurring in any execution of `AdjointPDR`. Instead, invariant (A1) informs us that the positive chain $\vec{x}$ is always in between the initial chain of $f \vee i$ and the final chain of $g \wedge p$. Such values of $\vec{x}$ are obtained by, respectively, simple initial and simple final heuristic. This is formally shown in Propositions 7.12 and 7.13 below.

Example 7.11. Consider the two runs of `AdjointPDR` in Example 7.3. The first one exploits the simple initial heuristic and indeed, the positive chain $\vec{x}$ coincides with the initial chain. Analogously, the second run uses the simple final heuristic. ■

Proposition 7.12. *Assume $p \neq \top$ and let $h\colon \mathcal{S} \to L$ be any simple initial heuristic. For all $(\vec{x}\|\vec{y})_{n,k} \in \mathcal{S}^h$, the first inequality in* (A1) *holds as an equality for all $j \in [0, n-2]$, namely $x_j = (f \vee i)^j(\bot)$.*

Proposition 7.13. *Assume $p \neq \top$ and let $h\colon \mathcal{S} \to L$ be any simple final heuristic. If $s_0 \to^* \xrightarrow{U} (\vec{x}\|\vec{y})_{n,k}$ then the second inequality in* (A1) *holds as an equality, namely for all $j \in [1, n-1]$, $x_j = (g \wedge p)^{n-1-j}(\top)$.*

7.3.5 Negative Termination

When the lattice L is not finite, `AdjointPDR` may not return a result, since checking $\mathrm{lfp}(f \vee i) \leq p$ is not always decidable. In this section, we show that the use of certain heuristics can guarantee termination whenever $\mathrm{lfp}(f \vee i) \not\leq p$. The key insight is the following: if $\mathrm{lfp}(f \vee i) \not\leq p$ then by (2.3), there should exist some $\tilde{n} \in \mathbb{N}$ such that $(f \vee i)^{\tilde{n}}(\bot) \not\leq p$. By (A1), the rule (Unfold) can be applied only when $(f \vee i)^{n-1}(\bot) \leq x_{n-1} \leq p$. Since (Unfold) increases n and n is never decreased by other rules, then (Unfold) can be applied at most $\tilde{n}$ times. Therefore, we can guarantee termination whenever the number of steps between two (Unfold) is finite.

The first observation for termination is the following lemma. It states that an element z cannot be added twice to negative sequence until n is increased, i.e., until (Unfold) is applied.

Lemma 7.14. *If* $s_0 \to^* s \overset{D}{\to}_z \to^* s' \overset{D}{\to}_{z'}$ *and* s *and* s' *carry the same index* (n,k)*, then* $z' \neq z$*. Similarly, if* $s_0 \to^* s \overset{Ca}{\to}_z \to^* s' \overset{Ca}{\to}_{z'}$ *and* s *and* s' *carry the same index* (n,k)*, then* $z' \neq z$*.*

Elements of negative sequences are introduced by rules (Candidate) and (Decide). If we guarantee that for any index (n,k) the heuristic in such cases returns a finite number of values for z, then we can prove termination. To make this formal, we fix $\mathit{CaD}^h_{n,k} \triangleq \{(\vec{x}\|\vec{y})_{n,k} \in \mathcal{S}^h \mid (\vec{x}\|\vec{y})_{n,k} \overset{Ca}{\to}$ or $(\vec{x}\|\vec{y})_{n,k} \overset{D}{\to}\}$, i.e., the set of all (n,k)-indexed states reachable by $\texttt{AdjointPDR}_h$ that trigger (Candidate) or (Decide), and $h(\mathit{CaD}^h_{n,k}) \triangleq \{h(s) \mid s \in \mathit{CaD}^h_{n,k}\}$, i.e., the set of all possible values returned by h in such states.

Theorem 7.15 (Negative termination)**.** *Let* h *be a legit heuristic. If* $h(\mathit{CaD}^h_{n,k})$ *is finite for all* n, k *and* $\mathit{lfp}(f \vee i) \not\leq p$*, then* $\texttt{AdjointPDR}_h$ *terminates.*

Corollary 7.16. *Let* h *be a simple heuristic. If* $\mathit{lfp}(f \vee i) \not\leq p$*, then* $\texttt{AdjointPDR}_h$ *terminates.*

Note that this corollary ensures negative termination whenever we use the canonical choices in (Candidate) and (Decide) *irrespective of the choice for* (Conflict), therefore it holds for both simple initial and simple final heuristics.

7.3.6 The meet-semilattice of positive chains and the join-semilattice of negative sequences

We conclude this section with two results illustrating some algebraic properties of positive chains and negative sequences. These are not necessary for proving properties of $\texttt{AdjointPDR}$, but they will be quite convenient in Section 7.4.2.

We observe that positive chains of a fixed length n form a join-semilattice and negative sequences a meet-semilattices, where joins and meets are defined point-wise, i.e., for two positive chains $\vec{x^1}, \vec{x^2}$ their join is defined as $(\vec{x^1} \vee \vec{x^2})_j \triangleq x^1_j \vee x^2_j$, and similarly for negative sequences. To show this it suffices to prove that the join of an arbitrary set of positive chains (resp. the meet of an arbitrary set of negative sequences) is still a positive chain (resp. negative sequence).

Lemma 7.17. *Let* I *be a set. For all* $m \in I$*, let* $\vec{x^m} = x^m_0, \dots, x^m_{n-1}$ *be a positive chain. Then, the chain* $\bigvee_{m \in I} \vec{x^m}$ *defined for all* $j \in [0, n-1]$ *as*

$$(\bigvee_{m \in I} \vec{x^m})_j \triangleq \bigvee_{m \in I} x^m_j$$

is a positive chain.

Proof. Since $i \leq x^m_1$ for all $m \in I$, then $i \leq \bigvee_{m \in I} x^m_1$. Since $x^m_{n-2} \leq p$ for all $m \in I$, then $\bigvee_{m \in I} x^m_{n-2} \leq p$.

To show that $f((\bigvee_{m\in I}\vec{x^m})_j) \leq (\bigvee_{m\in I}\vec{x^m})_{j+1}$ we just observe the following

$$\begin{aligned}
f((\bigvee_{m\in I}\vec{x^m})_j) &= f(\bigvee_{m\in I} x^m_j) && \text{[def.]}\\
&= \bigvee_{m\in I} f(x^m_j) && [f \dashv g]\\
&\leq \bigvee_{m\in I} x^m_{j+1} && \text{[(P3)]}\\
&= (\bigvee_{m\in I}\vec{x^m})_{j+1} && \text{[def.]}
\end{aligned}$$

Thus (P1), (P2) and (P3) hold for $\bigvee_{m\in I}\vec{x^m}$. □

Lemma 7.18. *Let I be a set. For all $m \in I$, let $\vec{y^m} = y^m_k, \ldots, y^m_{n-1}$ be a negative sequence. Then, the sequence $\bigwedge_{m\in I}\vec{y^m}$ defined for all $j = 0, \ldots n-1$ as*

$$(\bigwedge_{m\in I}\vec{y^m})_j \triangleq \bigwedge_{m\in I} y^m_j$$

is a negative sequence. Moreover, if $\vec{y^m}$ is conclusive for all $m \in I$, then also $\bigwedge_{m\in I}\vec{y^m}$ is conclusive.

Proof. Since $p \leq y^m_{n-1}$ for all $m \in I$, then $p \leq \bigwedge_{m\in I} y^m_{n-1}$.

To show that $g(\bigwedge_{m\in I}\vec{y^m})_{j+1} \leq (\bigwedge_{m\in I}\vec{y^m})_j$ we proceed as follows

$$\begin{aligned}
g(\bigwedge_{m\in I}\vec{y^m})_{j+1} &= g(\bigwedge_{m\in I} y^m_j) && \text{[def.]}\\
&= \bigwedge_{m\in I} g(y^m_{j+1}) && [f \dashv g]\\
&\leq \bigwedge_{m\in I} y^m_j && \text{[(N2)]}\\
&= (\bigwedge_{m\in I}\vec{y^m})_j && \text{[def.]}
\end{aligned}$$

For conclusive sequences, observe that, since $i \not\leq y^m_1$ for all $m \in I$, then $i \not\leq \bigwedge_{m\in I} y^m_1 = (\bigwedge_{m\in I}\vec{y^m})_1$. □

The bottom element of the meet-semilattice of negative sequences is given by $g^{n-1-j}(p)$ for all $j \in [k, n-1]$, and is exactly the one in invariant (A3). The top element of the join-semilattice of positive chains is the chain defined as $(g \wedge p)^{n-1-j}(\top)$ for all $j \in [0, n-1]$; its bottom element is the chain $(f \vee i)^j(\bot)$. Again, these are exactly the bounds that appear in invariant (A1). Note that, if $\vec{y^1}$ and $\vec{y^2}$ are conclusive, also $\vec{y^1} \wedge \vec{y^2}$ is conclusive. An analogous property for positive chains does not hold.

7.4 AdjointPDR$^{\downarrow}$

In Section 7.2, we have introduced an algorithm for checking $\mathrm{lfp}(b) \leq p$ whenever b is of the form $f \vee i$ for an element $i \in L$ and a left-adjoint $f \colon L \to L$. This, unfortunately, is not the case for several interesting problems, like the max reachability problem for Markov Decision Processes [BK08] that we will illustrate in Section 7.5.

The next result informs us that, under standard assumptions, one can transfer the problem of checking $\mathrm{lfp}(b) \leq p$ to lower sets, where adjoints can always be defined. Recall that, for a lattice $(L, \leq)$, a *lower set* is a subset $X \subseteq L$ such that if $x \in X$ and $x' \leq x$ then $x' \in X$; the set of lower sets of L forms a complete lattice $(L^{\downarrow}, \subseteq)$ with joins and meets given by union and intersection; as expected $\bot$ is $\emptyset$ and $\top$ is L. Given $b\colon L \to L$, one can define two functions $b^{\downarrow}, b_r^{\downarrow}\colon L^{\downarrow} \to L^{\downarrow}$ as $b^{\downarrow}(X) \triangleq b(X)^{\downarrow}$ and $b_r^{\downarrow}(X) \triangleq \{x \mid b(x) \in X\}$. It holds that $b^{\downarrow} \dashv b_r^{\downarrow}$.

$$\begin{tikzcd} (L,\leq) \arrow[loop left, "b"] \arrow[r, bend right, "(-)^{\downarrow}"'] \arrow[r, phantom, "\perp"] & (L^{\downarrow},\subseteq) \arrow[l, bend right, "\bigsqcup"'] \arrow[loop right, "b^{\downarrow} \dashv b_r^{\downarrow}"] \end{tikzcd} \tag{7.3}$$

In the diagram above, $(-)^{\downarrow}\colon x \mapsto \{x' \mid x' \leq x\}$ and $\bigsqcup\colon L^{\downarrow} \to L$ maps a lower set X into $\bigsqcup\{x \mid x \in X\}$. The maps $\bigsqcup$ and $(-)^{\downarrow}$ form a *Galois insertion*, namely $\bigsqcup \dashv (-)^{\downarrow}$ and $\bigsqcup(-)^{\downarrow} = id$, and thus one can think of (7.3) in terms of abstract interpretation: $L^{\downarrow}$ represents the concrete domain, L the abstract domain and b is a sound abstraction of $b^{\downarrow}$. Moreover, b is *forward-complete* [GRS00; BGGP18] w.r.t. $b^{\downarrow}$, namely:

$$(-)^{\downarrow} \circ b = b^{\downarrow} \circ (-)^{\downarrow} \tag{7.4}$$

Proposition 7.19. *Let $(L, \leq)$ be a complete lattice, $p \in L$ and $b\colon L \to L$ be a ω-continuous map. Then lfp$(b) \leq p$ iff lfp$(b^{\downarrow} \cup \bot^{\downarrow}) \subseteq p^{\downarrow}$.*

Proof. A simple inductive argument using (7.4) confirms that

$$(b^n x)^{\downarrow} = (b^{\downarrow})^n x^{\downarrow} \tag{7.5}$$

for all $x \in L$. The following sequence of logical equivalences

$$\begin{aligned}
\mathrm{lfp}(b) \leq p &\Leftrightarrow \forall n \in \mathbb{N}.\ b^n \bot \leq p && [\text{Theorem 2.3}]\\
&\Leftrightarrow \forall n \in \mathbb{N}.\ (b^n \bot)^{\downarrow} \subseteq p^{\downarrow} && [\text{mon. of } (-)^{\downarrow}, \bigsqcup \text{ and } \bigsqcup(-)^{\downarrow} = id]\\
&\Leftrightarrow \bigcup_{n \in \mathbb{N}} (b^n \bot)^{\downarrow} \subseteq p^{\downarrow} && [\text{def. of } \bigcup]\\
&\Leftrightarrow \bigcup_{n \in \mathbb{N}} (b^{\downarrow})^n \bot^{\downarrow} \subseteq p^{\downarrow} && [(7.5)]\\
&\Leftrightarrow \mathrm{lfp}(b^{\downarrow} \cup \bot^{\downarrow}) \subseteq p^{\downarrow} && [\text{Theorem 2.3}].
\end{aligned}$$

concludes the proof of the main statement. □

By means of Proposition 7.19, we can thus solve $\mathrm{lfp}(b) \leq p$ in L by running `AdjointPDR` on $(\bot^{\downarrow}, b^{\downarrow}, b_r^{\downarrow}, p^{\downarrow})$. Hereafter, we tacitly assume that b is ω-continuous.

7.4.1 `AdjointPDR`$^{\downarrow}$: Positive Chain in L, Negative Sequence in $L^{\downarrow}$

While `AdjointPDR` on $(\bot^{\downarrow}, b^{\downarrow}, b_r^{\downarrow}, p^{\downarrow})$ might be computationally expensive, it is the first step toward an efficient algorithm that exploits a convenient form of the positive chain.

A lower set $X \in L^{\downarrow}$ is said to be a *principal* if $X = x^{\downarrow}$ for some $x \in L$. Observe that the top of the lattice $(L^{\downarrow}, \subseteq)$ is a principal, namely $\top^{\downarrow}$, and that the meet (intersection) of two principals $x^{\downarrow}$ and $y^{\downarrow}$ is the principal $(x \wedge y)^{\downarrow}$.

Suppose that, in (Conflict), `AdjointPDR`$(\bot^{\downarrow}, b^{\downarrow}, b_r^{\downarrow}, p^{\downarrow})$ always chooses principals rather than arbitrary lower sets. This suffices to guarantee that all the elements of $\vec{x}$ are principals (with the only exception of x_0 which is constantly the bottom element of $L^{\downarrow}$ that, note,

```
                         AdjointPDR$^{\downarrow}$ $(b,p)$
<INITIALISATION>
  $(\vec{x}\|\vec{Y})_{n,k}$ := $(\emptyset,\bot,\top\|\varepsilon)_{3,3}$
<ITERATION>
  case $(\vec{x}\|\vec{Y})_{n,k}$ of                  % $\vec{x},\vec{Y}$ not conclusive
      $\vec{Y}=\varepsilon$ and $x_{n-1}\leq p$ :                %(Unfold)
          $(\vec{x}\|\vec{Y})_{n,k}$ := $(\vec{x},\top\|\varepsilon)_{n+1,n+1}$
      $\vec{Y}=\varepsilon$ and $x_{n-1}\not\leq p$ :                %(Candidate)
          choose $Z\in L^{\downarrow}$ such that $x_{n-1}\notin Z$ and $p\in Z$;
          $(\vec{x}\|\vec{Y})_{n,k}$ := $(\vec{x}\|Z)_{n,n-1}$
      $\vec{Y}\neq\varepsilon$ and $b(x_{k-1})\notin Y_k$ :           %(Decide)
          choose $Z\in L^{\downarrow}$ such that $x_{k-1}\notin Z$ and $b_r^{\downarrow}(Y_k)\subseteq Z$;
          $(\vec{x}\|\vec{Y})_{n,k}$ := $(\vec{x}\|Z,\vec{Y})_{n,k-1}$
      $\vec{Y}\neq\varepsilon$ and $b(x_{k-1})\in Y_k$ :           %(Conflict)
          choose $z\in L$ such that $z\in Y_k$ and $b(x_{k-1}\wedge z)\leq z$;
          $(\vec{x}\|\vec{Y})_{n,k}$ := $(\vec{x}\wedge_k z\|\mathsf{tail}(\vec{Y}))_{n,k+1}$
  endcase
<TERMINATION>
   if $\exists j\in[0,n-2]\,.\,x_{j+1}\leq x_j$ then return true   % $\vec{x}$ conclusive
   if $Y_1=\emptyset$ then return false                % $\vec{Y}$ conclusive
```

Figure 7.3: The algorithm AdjointPDR$^{\downarrow}$ for checking $\mathrm{lfp}(b)\leq p$: the elements of negative sequence are in $L^{\downarrow}$, while those of the positive chain are in L, with the only exception of x_0 which is constantly the bottom lower set $\emptyset$. For x_0, we fix $b(x_0)=\bot$.

is $\emptyset$ and not $\bot^{\downarrow}$). In fact, the elements of $\vec{x}$ are all obtained by (Unfold), that adds the principal $\top^{\downarrow}$, and by (Conflict), that takes their meets with the chosen principal.

Since principals are in bijective correspondence with the elements of L, by imposing to AdjointPDR$(\bot^{\downarrow}, b^{\downarrow}, b_r^{\downarrow}, p^{\downarrow})$ to choose a principal in (Conflict), we obtain an algorithm, named AdjointPDR$^{\downarrow}$, where the elements of the positive chain are drawn from L, while the negative sequence is taken in $L^{\downarrow}$. The algorithm is reported in Figure 7.3 where we use the notation $(\vec{x}\|\vec{Y})_{n,k}$ to emphasize that the elements of the negative sequence are lower sets of elements in L.

All definitions and results illustrated in Sections 7.2 and 7.3 for AdjointPDR are inherited by AdjointPDR$^{\downarrow}$, with the only exception of Proposition 7.6.3. This does not hold because it prescribes a choice for (Conflict) that may not be a principal. In contrast, the choice in Proposition 7.6.4 is, thanks to (7.4), a principal. This means in particular that the simple initial heuristic is always applicable.

Theorem 7.20. *All results in Section 7.3, but Proposition 7.6.3, hold for* AdjointPDR$^{\downarrow}$.

7.4.2 AdjointPDR$^{\downarrow}$ simulates LT-PDR

The closest approach to AdjointPDR and AdjointPDR$^{\downarrow}$ is the lattice-theoretic extension of the original PDR, called LT-PDR [Kor+22]. While these algorithms exploit essentially the same positive chain to find an invariant, the main difference lies in the sequence used to witness the existence of some counterexamples.

Definition 7.21 (Kleene sequence, from [Kor+22])**.** A sequence $\vec{c} = c_k, \dots, c_{n-1}$ of elements of L is a *Kleene sequence* if the conditions *(C1)* and *(C2)* below hold. It is *conclusive* if also condition *(C0)* holds.

$$\textit{(C0)}\ c_1 \leq b(\bot), \qquad \textit{(C1)}\ c_{n-1} \not\leq p, \qquad \textit{(C2)}\ \forall j \in [k, n-2].\ c_{j+1} \leq b(c_j).$$

LT-PDR tries to construct an under-approximation c_{n-1} of $b^{n-2}(\bot)$ that violates the property p. The Kleene sequence is constructed by trial and error, starting by some arbitrary choice of c_{n-1}.

`AdjointPDR` crucially differs from LT-PDR in the search for counterexamples: LT-PDR under-approximates the final chain while `AdjointPDR` over-approximates it. However, we can draw a formal correspondence between `AdjointPDR`$^{\downarrow}$ and LT-PDR by showing that `AdjointPDR`$^{\downarrow}$ simulates LT-PDR, but cannot be simulated by LT-PDR. In fact, `AdjointPDR`$^{\downarrow}$ exploits the existence of the adjoint to start from an over-approximation Y_{n-1} of $p^{\downarrow}$ and computes backward an over-approximation of the set of safe states. Thus, the key difference comes from the strategy to look for a counterexample: to prove $\mathrm{lfp}(b) \not\leq p$, `AdjointPDR`$^{\downarrow}$ tries to find Y_{n-1} satisfying $p \in Y_{n-1}$ and $\mathrm{lfp}(b) \notin Y_{n-1}$ while LT-PDR tries to find c_{n-1} s.t. $c_{n-1} \not\leq p$ and $c_{n-1} \leq \mathrm{lfp}(b)$.

Theorem 7.23 below states that `AdjointPDR`$^{\downarrow}$ can mimic any execution of LT-PDR. The proof exploits a map from LT-PDR's Kleene sequences $\vec{c}$ to `AdjointPDR`$^{\downarrow}$'s negative sequences $\mathit{neg}(\vec{c})$ of a particular form. Let $(L^{\uparrow}, \supseteq)$ be the complete lattice of upper sets, namely subsets $X \subseteq L$ such that $X = X^{\uparrow} \triangleq \{x' \in L \mid \exists x \in X \,.\, x \leq x'\}$. There is an isomorphism $\neg\colon (L^{\uparrow}, \supseteq) \xleftrightarrow{\cong} (L^{\downarrow}, \subseteq)$ mapping each $X \subseteq S$ into its complement. For a Kleene sequence $\vec{c} = c_k, \dots, c_{n-1}$ of LT-PDR, the sequence $\mathit{neg}(\vec{c}) \triangleq \neg(\{c_k\}^{\uparrow}), \dots, \neg(\{c_{n-1}\}^{\uparrow})$ is a negative sequence, in the sense of Definition 7.2, for `AdjointPDR`$^{\downarrow}$.

Proposition 7.22. *Let $\vec{c}$ be a Kleene sequence. Then $\mathit{neg}(\vec{c})$ is a negative sequence for* `AdjointPDR`$^{\downarrow}$*.*

Proof. First, we show that $p \in \mathit{neg}(\vec{c})_{n-1}$. Since $c_{n-1} \not\leq p$, by (C1), then $p \notin \{c_{n-1}\}^{\uparrow}$. Thus $p \in \neg(\{c_{n-1}\}^{\uparrow})$, that is $p \in \mathit{neg}(\vec{c})_{n-1}$.

Then, we show that $b_r^{\downarrow}(\mathit{neg}(\vec{c})_{j+1}) \subseteq \mathit{neg}(\vec{c})_j$.

$$\begin{aligned}
b_r^{\downarrow}(\mathit{neg}(\vec{c})_{j+1}) &= b_r^{\downarrow}(\neg(\{c_{j+1}\}^{\uparrow})) && \text{[def.]}\\
&= \{x \mid b(x) \notin (\{c_{j+1}\}^{\uparrow}\} && \text{[def.]}\\
&= \{x \mid c_{j+1} \not\leq b(x)\} && \text{[def.]}\\
&\subseteq \{x \mid b(c_j) \not\leq b(x)\} && \text{[(C2)]}\\
&\subseteq \{x \mid c_j \not\leq x\} && \text{[mon. of } b\text{]}\\
&= \neg(\{c_j\}^{\uparrow}) && \text{[def.]}\\
&= \mathit{neg}(\vec{c})_j && \text{[def.]}
\end{aligned}$$

□

Most importantly, the assignment $\vec{c} \mapsto \mathit{neg}(\vec{c})$ extends to a function, from the states of LT-PDR to those of `AdjointPDR`$^{\downarrow}$, that is proved to be a *strong simulation* [Mil89].

Theorem 7.23. `AdjointPDR`$^{\downarrow}$ *simulates LT-PDR.*

Remarkably, `AdjointPDR`$^{\downarrow}$'s negative sequences are not limited to the images of LT-PDR's Kleene sequences: they are more general than the complement of the upper closure of a singleton. In fact, a single negative sequence of `AdjointPDR`$^{\downarrow}$ can represent *multiple* Kleene sequences of LT-PDR at once. Intuitively, this means that a single execution of `AdjointPDR`$^{\downarrow}$ can correspond to multiple runs of LT-PDR. We can make this formal by means of the following result.

Proposition 7.24. *Let $\{\vec{c^m}\}_{m\in M}$ be a family of Kleene sequences. Then its pointwise intersection $\bigcap_{m\in M} neg(\vec{c^m})$ is a negative sequence.*

Proof. Since each of the $\vec{c^m}$ is a Kleene sequence, for all $m \in M$, $neg(\vec{c^m})$ is, by Proposition 7.22, a negative sequence. Since negative sequences form a meet-semilattice, their intersection is also a negative sequence (Lemma 7.18). □

The above intersection is pointwise in the sense that, for all $j \in [k, n-1]$, it holds $(\bigcap_{m\in M} neg(\vec{c^m}))_j \triangleq \bigcap_{m\in M}(neg(\vec{c^m}))_j = \neg(\{c_j^m \mid m \in M\}^{\uparrow})$: intuitively, this is (up to $neg(\cdot)$) a set containing all the M counterexamples. Note that, if the negative sequence of `AdjointPDR`$^{\downarrow}$ makes (A3) hold as an equality, as it is possible with any simple heuristic (see Proposition 7.10), then its complement contains *all* Kleene sequences possibly computed by LT-PDR.

Proposition 7.25. *Let $\vec{c}$ be a Kleene sequence and $\vec{Y}$ be the negative sequence s.t. $Y_j = (b_r^{\downarrow})^{n-1-j}(p^{\downarrow})$ for all $j \in [k, n-1]$. Then $c_j \in \neg(Y_j)$ for all $j \in [k, n-1]$.*

Proof. Since $\vec{c} = c_0, \ldots, c_{n-1}$ is a Kleene sequence, $neg(\vec{c}) = \neg(\{c_k\}^{\uparrow}), \ldots, \neg(\{c_{n-1}\}^{\uparrow})$ is, by Proposition 7.22, a negative sequence. By (A3), for all $j \in [k, n-1]$ we have $(b_r^{\downarrow})^{n-1-j}(p^{\downarrow}) \subseteq \neg(\{c_j\}^{\uparrow})$. Therefore, $\neg(b_r^{\downarrow})^{n-1-j}(p^{\downarrow}) \supseteq \{c_j\}^{\uparrow}$, so $c_j \in \neg(b_r^{\downarrow})^{n-1-j}(p^{\downarrow}) = \neg(Y_j)$. □

While the previous result suggests that simple heuristics are always the best in theory, as they can carry all counterexamples, this is often not the case in practice, since they might be computationally hard and outperformed by some smart over-approximations. An example is given by (7.7) in the next section.

7.5 Instantiating `AdjointPDR`$^{\downarrow}$ for MDPs

In this section we illustrate how to use `AdjointPDR`$^{\downarrow}$ to address the max reachability problem [BK08] for Markov Decision Processes.

7.5.1 The max reachability problem

A *Markov Decision Process* (MDP) is a tuple (A, S, s_ι, δ) where A is a set of labels, S is a set of states, $s_\iota \in S$ is an initial state, and $\delta\colon S \times A \to \mathcal{D}S + 1$ is a transition function. Here $\mathcal{D}S$ is the set of probability distributions over S, namely functions $d\colon S \to [0,1]$ such that $\sum_{s\in S} d(s) = 1$, and $\mathcal{D}S + 1$ is the disjoint union of $\mathcal{D}S$ and $1 = \{*\}$. The transition function δ assigns to every label $a \in A$ and to every state $s \in S$ either a distribution of states or $* \in 1$. We assume that both S and A are finite sets and that the set $Act(s) \triangleq \{a \in A \mid \delta(s, a) \neq *\}$ of actions enabled at s is non-empty for all states.

An MDP (A, S, s_ι, δ) mixes nondeterministic and probabilistic computations. The notion of a *scheduler*, also known as *adversary*, *policy* or *strategy*, is used to resolve nondeterministic choices. Below, we write S^+ for the set of non-empty sequence over S, intuitively representing runs of the MDP. A scheduler is a function $\alpha\colon S^+ \to A$ such that $\alpha(s_0 s_1 \ldots s_n) \in Act(s_n)$: given the states visited so far, the scheduler decides which action to trigger among the enabled ones so that the MDP behaves as a Markov chain. A scheduler α is called *memoryless* if it always selects the same action in a given state, namely, if $\alpha(s_0 s_1 \dots s_n) = \alpha(s_n)$ for any sequence $s_0 s_1 \dots s_n \in S^+$. Memoryless schedulers can thus be represented just as functions $\alpha\colon S \to A$ such that $\alpha(s) \in \mathit{Act}(s)$ for any $s \in S$.

Given an MDP, the *max reachability problem* requires to check whether the probability of reaching some bad states $\beta \subseteq S$ is less than or equal to a given threshold $\lambda \in [0,1]$ for all possible schedulers. Thus, to solve this problem, one should compute the supremum over infinitely many schedulers. Notably, it is known that there always exists one memoryless scheduler that maximizes the probabilities to reach β (see e.g. [BK08]). As the memoryless schedulers are finitely many (although their number can grow exponentially), the supremum can thus be replaced by a maximum.

7.5.2 Applying `AdjointPDR`$^{\downarrow}$ to the max reachability problem

The max-reachability problem, namely checking for a MDP (A, S, s_ι, δ) whether the probability of reaching some bad states $\beta \subseteq P$ is less than a threshold $\lambda \in [0,1]$, enjoys a convenient lattice-theoretical formulation [BK08].

Consider the lattice $([0,1]^S, \leq)$ of all functions $d\colon S \to [0,1]$, often called frames or fuzzy predicates, ordered pointwise. The max reachability problem is equivalent to check that $\mu b \leq p$ for $p \in [0,1]^S$ and $b\colon [0,1]^S \to [0,1]^S$, defined for all $d \in [0,1]^S$ and $s \in S$, as

$$p(s) \triangleq \begin{cases} \lambda & \text{if } s = s_\iota, \\ 1 & \text{if } s \neq s_\iota, \end{cases} \qquad b(d)(s) \triangleq \begin{cases} 1 & \text{if } s \in \beta, \\ \max\limits_{a \in \mathit{Act}(s)} \sum\limits_{s' \in S} d(s') \cdot \delta(s,a)(s') & \text{if } s \notin \beta. \end{cases}$$

Since b is not of the form $f \vee i$ for a left adjoint f (see e.g. [Kor+22]), we can't use `AdjointPDR`, but we can use `AdjointPDR`$^{\downarrow}$. Beyond the simple initial heuristic, which is always applicable and enjoys negative termination, we illustrate now two additional heuristics that are experimentally tested in Section 7.7.

The two novel heuristics make the same choices in (Candidate) and (Decide). They exploit memoryless schedulers $\alpha\colon S \to A$, and the function $b_\alpha\colon [0,1]^S \to [0,1]^S$ defined for all $d \in [0,1]^S$ and $s \in S$ as follows:

$$b_\alpha(d)(s) \triangleq \begin{cases} 1 & \text{if } s \in \beta, \\ \sum_{s' \in S} d(s') \cdot \delta(s, \alpha(s))(s') & \text{otherwise.} \end{cases} \tag{7.6}$$

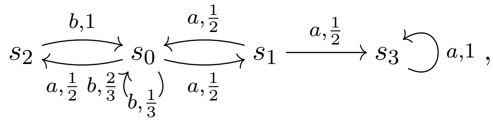
Since for all $D \in ([0,1]^S)^{\downarrow}$, $b_r^{\downarrow}(D) = \{d \mid b(d) \in D\} = \bigcap_\alpha \{d \mid b_\alpha(d) \in D\}$ and since `AdjointPDR`$^{\downarrow}$ executes (Decide) only when $b(x_{k-1}) \notin Y_k$, there should exist some α such that $b_\alpha(x_{k-1}) \notin Y_k$. One can thus fix

$$(\vec{x} \| \vec{Y})_{n,k} \mapsto \begin{cases} p^{\downarrow} & \text{if } (\vec{x} \| \vec{Y})_{n,k} \overset{Ca}{\to} \\ \{d \mid b_\alpha(d) \in Y_k\} & \text{if } (\vec{x} \| \vec{Y})_{n,k} \overset{D}{\to} \end{cases} \tag{7.7}$$

Intuitively, such choices are smart refinements of those in (7.2): for (Candidate) they are exactly the same; for (Decide) rather than taking $b_r^{\downarrow}(Y_k)$, we consider a larger lower-set determined by the labels chosen by α. This allows to represent each Y_j as a set of $d \in [0,1]^S$ satisfying a *single* linear inequality, while using $b_r^{\downarrow}(Y_k)$ would yield a systems of possibly exponentially many inequalities (see Example 7.27 below). Moreover, from Theorem 7.15, it follows that such choices ensures negative termination.

Corollary 7.26. *Let h be a legit heuristic defined for (Candidate) and (Decide) as in (7.7). If $\mu b \not\leq p$, then* $\texttt{AdjointPDR}^{\downarrow}{}_h$ *terminates.*

Example 7.27. Consider the maximum reachability problem with threshold $\lambda = \frac{1}{4}$ and $\beta = \{s_3\}$ for the following MDP on alphabet $A = \{a, b\}$ and $s_\iota = s_0$.



Hereafter we write $d \in [0,1]^S$ as column vectors with four entries $v_0 \dots v_3$ and we will use $\cdot$ for the usual matrix multiplication. With this notation, the lower set $p^{\downarrow} \in ([0,1]^S)^{\downarrow}$ and $b\colon [0,1]^S \to [0,1]^S$ can be written as

$$p^{\downarrow} = \{\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \mid \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix} \leq \begin{bmatrix} \frac{1}{4} \end{bmatrix}\} \quad \text{and} \quad b(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \max(\frac{v_1+v_2}{2}, \frac{v_0+2v_2}{3}) \\ \frac{v_0+v_3}{2} \\ v_0 \\ 1 \end{bmatrix}.$$

Amongst the several memoryless schedulers, only two are relevant for us:

$$\zeta \triangleq (s_0 \mapsto a,\ s_1 \mapsto a,\ s_2 \mapsto b,\ s_3 \mapsto a) \text{ and} \tag{7.8}$$

$$\xi \triangleq (s_0 \mapsto b,\ s_1 \mapsto a,\ s_2 \mapsto b,\ s_3 \mapsto a). \tag{7.9}$$

By using the definition of $b_\alpha\colon [0,1]^S \to [0,1]^S$ in (7.6), we have that

$$b_\zeta(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \frac{v_1+v_2}{2} \\ \frac{v_0+v_3}{2} \\ v_0 \\ 1 \end{bmatrix} \qquad \text{and} \qquad b_\xi(\begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{bmatrix}) = \begin{bmatrix} \frac{v_0+2v_2}{3} \\ \frac{v_0+v_3}{2} \\ v_0 \\ 1 \end{bmatrix}.$$

It is immediate to see that the problem has negative answer, since using ζ in 4 steps or less, s_0 can reach s_3 already with probability $\frac{1}{4} + \frac{1}{8}$.

To illustrate the advantages of (7.7), we run $\texttt{AdjointPDR}^{\downarrow}$ with the simple initial heuristic and with the heuristic that only differs for the choice in (Decide), taken as in (7.7). For both heuristics, the first iterations are the same: several repetitions of (Candidate), (Conflict) and (Unfold) exploiting elements of the positive chain that form the initial chain (except for the last element x_{n-1}).

$$(\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| \varepsilon)_{3,3} \overset{Ca}{\to}\overset{Co}{\to} (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \| \varepsilon)_{3,3} \overset{U}{\to}\overset{Ca}{\to}\overset{Co}{\to}\overset{U}{\to}\overset{Ca}{\to}\overset{Co}{\to}\overset{U}{\to}\overset{Ca}{\to}\overset{Co}{\to}\overset{U}{\to}\overset{Ca}{\to} (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{5}{8} \\ \frac{1}{4} \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| p^{\downarrow})_{7,6}.$$

In the latter state the algorithm has to perform (Decide), since $b(x_5) \notin p^{\downarrow}$. Now the choice of z in (Decide) is different for the two heuristics: the former uses $b_r^{\downarrow}(p^{\downarrow}) = \{d \mid b(d) \in p^{\downarrow}\}$, the latter uses $\{d \mid b_\zeta(d) \in p^{\downarrow}\}$. Despite the different choices, both the heuristics proceed with 6 steps of (Decide):

$$(\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{5}{8} \\ \frac{1}{4} \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| \mathcal{F}^0)_{7,6} \overset{D}{\to}\overset{D}{\to}\overset{D}{\to}\overset{D}{\to}\overset{D}{\to} (\emptyset \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{1}{2} \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} \frac{1}{4} \\ \frac{5}{8} \\ \frac{1}{4} \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \| \mathcal{F}^5, \mathcal{F}^4, \mathcal{F}^3, \mathcal{F}^2, \mathcal{F}^1, \mathcal{F}^0)_{7,1}$$

The element of the negative sequence $\mathcal{F}^i$ are illustrated in Figure 7.4 for both the heuristics. In both cases, $\mathcal{F}^5 = \emptyset$ and thus $\texttt{AdjointPDR}^{\downarrow}$ returns false.

To appreciate the advantages provided by (7.7), it is enough to compare the two columns for the $\mathcal{F}^i$ in Figure 7.4: in the central column, the number of inequalities defining $\mathcal{F}^i$ grows significantly, while in the rightmost column is always 1. ■

$$\begin{array}{ll|c|c}
\mathcal{F}^0 & \triangleq & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\} & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\} \\
\mathcal{F}^1 & \triangleq & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0 & 1 & 1 & 0\\1 & 0 & 2 & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{2}\\\frac{3}{4}\end{bmatrix}\} & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0 & \frac{1}{2} & \frac{1}{2} & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\} \\
\mathcal{F}^2 & \triangleq & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}3 & 0 & 0 & 1\\2 & 1 & 1 & 0\\4 & 0 & 2 & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}1\\\frac{3}{2}\\\frac{9}{4}\end{bmatrix}\} & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}\frac{3}{4} & 0 & 0 & \frac{1}{4}\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}\frac{1}{4}\end{bmatrix}\} \\
\mathcal{F}^3 & \triangleq & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0 & \frac{3}{2} & \frac{3}{2} & 0\\1 & 0 & 2 & 0\\\frac{3}{2} & 1 & 1 & \frac{1}{2}\\\frac{13}{6} & 0 & \frac{4}{3} & \frac{1}{2}\\2 & 2 & 2 & 0\\\frac{10}{3} & 0 & \frac{8}{3} & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\\0\\\frac{3}{2}\\\frac{3}{2}\\\frac{9}{4}\\\frac{9}{4}\end{bmatrix}\} & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0 & \frac{3}{8} & \frac{3}{8} & 0\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\end{bmatrix}\} \\
\mathcal{F}^4 & \triangleq & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\0 & 0 & 0 & 1\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\\0\\0\\0\end{bmatrix}\} = \{\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\} & \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}\frac{9}{16} & 0 & 0 & \frac{3}{16}\end{bmatrix}\cdot\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq \begin{bmatrix}0\end{bmatrix}\} \\
\mathcal{F}^5 & \triangleq & \emptyset & \emptyset
\end{array}$$

Figure 7.4: The elements of the negative sequences computed by AdjointPDR$^{\downarrow}$ for the MDP in Example 7.27. In the central column, these elements are computed by means of the simple initial heuristics, that is $\mathcal{F}^i = (b_r^{\downarrow})^i(p^{\downarrow})$. In the rightmost column, these elements are computed using the heuristic in (7.7). In particular $\mathcal{F}^i = \{d \mid b_\zeta(d) \in \mathcal{F}^{i-1}\}$ for $i \leq 3$, while for $i \geq 4$ these are computed as $\mathcal{F}^i = \{d \mid b_\xi(d) \in \mathcal{F}^{i-1}\}$.

The fact that using (7.7) ensures that Y_k is generated by a single linear inequality is quite convenient. Indeed, in this case

$$Y_k = \{d \in [0,1]^S \mid \sum_{s \in S}(r_s \cdot d(s)) \leq r\}$$

for suitable non-negative real numbers r and r_s for all $s \in S$. The convex set Y_k is generated by finitely many $d \in [0,1]^S$ enjoying a useful property: $d(s)$ is different from 0 and 1 only for at most one $s \in S$. The set of its generators, denoted by $\mathcal{G}_k$, can thus be easily computed. We exploit this property to resolve the choice for (Conflict). We consider its subset $\mathcal{Z}_k \triangleq \{d \in \mathcal{G}_k \mid b(x_{k-1}) \leq d\}$ and define $z_B, z_{01} \in [0,1]^S$ for all $s \in S$ as

$$z_B(s) \triangleq \begin{cases} (\bigwedge \mathcal{Z}_k)(s) & \text{if } r_s \neq 0,\ \mathcal{Z}_k \neq \emptyset \\ b(x_{k-1})(s) & \text{otherwise} \end{cases} \quad z_{01}(s) \triangleq \begin{cases} \lceil z_B(s) \rceil & \text{if } r_s = 0, \mathcal{Z}_k \neq \emptyset \\ z_B(s) & \text{otherwise} \end{cases} \tag{7.10}$$

where, for $u \in [0,1]$, $\lceil u \rceil$ denotes 0 if $u = 0$ and 1 otherwise. We call hCoB and hCo01 the heuristics defined as in (7.7) for (Candidate) and (Decide) and as z_B, respectively z_{01}, for (Conflict). The heuristics hCo01 can be seen as a Boolean modification of hCoB, rounding up positive values to 1 to accelerate convergence.

Proposition 7.28. *The heuristics* hCoB *and* hCo01 *are legit.*

By Corollary 7.26, AdjointPDR$^{\downarrow}$ terminates for negative answers with both hCoB and hCo01. We conclude this section with a last example.

$$(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\varepsilon)_{2,2} \overset{Ca}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{2,1} \overset{Co}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\|\varepsilon)_{2,2}$$

$$\overset{U}{\to}\overset{Ca}{\to}(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{3,2} \overset{Co}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\|\varepsilon)_{3,3}$$

$$\overset{U}{\to}\overset{Ca}{\to}(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{4,3} \overset{Co}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\|\varepsilon)_{4,4}$$

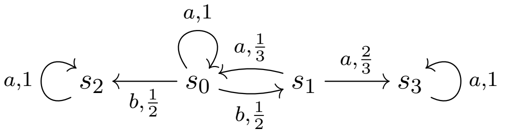
$$\overset{U}{\to}\overset{Ca}{\to}(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{5,4} \overset{Co}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{7}{9}\\0\\1\end{bmatrix}\|\varepsilon)_{4,4}$$

$$\overset{U}{\to}\overset{Ca}{\to}(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{7}{9}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{6,5} \overset{Co}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{7}{9}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{7}{18}\\\frac{7}{9}\\0\\1\end{bmatrix}\|\varepsilon)_{6,6}$$

$$\overset{U}{\to}\overset{Ca}{\to}(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{7}{9}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{7}{18}\\\frac{7}{9}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{7,6} \overset{Co}{\to} (\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{2}{3}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{1}{3}\\\frac{7}{9}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{7}{18}\\\frac{7}{9}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{7}{18}\\\frac{43}{54}\\0\\1\end{bmatrix}\|\varepsilon)_{7,7}\cdots$$

Figure 7.5: The non-terminating execution of `AdjointPDR`$^{\downarrow}$ with the simple initial heuristics for the max reachability problem of Example 7.29. The elements of the positive chain, with the exception of the last one x_{n-1} are those of the initial chain.

Example 7.29. Consider the following MDP with alphabet $A = \{a, b\}$ and $s_\iota = s_0$



and the max reachability problem with threshold $\lambda = \frac{2}{5}$ and $\beta = \{s_3\}$. The lower set $p^{\downarrow} \in ([0,1]^S)^{\downarrow}$ and $b\colon [0,1]^S \to [0,1]^S$ can be written as

$$p^{\downarrow} = \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}1 & 0 & 0 & 0\end{bmatrix} \cdot \begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq [\tfrac{2}{5}]\} \quad \text{and} \quad b(\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix}) = \begin{bmatrix}\max(v_0, \frac{v_1+v_2}{2})\\ \frac{v_0+2\cdot v_3}{3}\\ v_2\\ 1\end{bmatrix}$$

Consider also the scheduler $\xi\colon S \to A$ defined as $\xi \triangleq [s_0 \mapsto b, s_1 \mapsto a, s_2 \mapsto a, s_3 \mapsto a]$, for which we illustrate

$$b_\xi(\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix}) = \begin{bmatrix}\frac{v_1+v_2}{2}\\ \frac{v_0+2\cdot v_3}{3}\\ v_2\\ 1\end{bmatrix} \quad \text{and} \quad \mathcal{F}_\xi^1 \triangleq \{d \mid b_\xi(d) \in p^{\downarrow}\} = \{\begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \mid \begin{bmatrix}0 & 1 & 1 & 0\end{bmatrix} \cdot \begin{bmatrix}v_0\\v_1\\v_2\\v_3\end{bmatrix} \leq [\tfrac{4}{5}]\} \tag{7.11}$$

With the simple initial heuristic, `AdjointPDR`$^{\downarrow}$ does not terminate (Figure 7.5). With the heuristic `hCo01` using scheduler ξ, it returns true in 14 steps (Figure 7.6), while with `hCoB` in 8 (Figure 7.7). The first 4 steps of both `hCoB` and `hCo01` are the same: in the first (Conflict) $z_B = z_{01}$, while in the second $z_{01}(s_1) = 1$ and $z_B(s_1) = \frac{4}{5}$, leading to the two different executions. Observe that this difference is due to the fact that in $p^{\downarrow}$, the coefficient corresponding to s_1, namely r_{s_1}, is 0 and, since $\mathcal{Z}_3 \neq \emptyset$, then $z_{01}(s_1) = \lceil z_B(s_1) \rceil$. ■

$$(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\varepsilon)_{3,3}\xrightarrow{Ca}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{3,2}\xrightarrow{Co}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\|\varepsilon)_{3,3}$$

$$\xrightarrow{U}\xrightarrow{Ca}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{4,3}\xrightarrow{Co}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix}\|\varepsilon)_{4,4}$$

$$\xrightarrow{U}\xrightarrow{Ca}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{5,4}\xrightarrow{D}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\mathcal{F}_{\xi}^{1}p^{\downarrow})_{5,3}$$

$$\xrightarrow{Co}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{5,4}$$

$$\xrightarrow{Co}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix}\|\varepsilon)_{5,5}$$

$$\xrightarrow{U}\xrightarrow{Ca}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{6,5}\xrightarrow{D}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\mathcal{F}_{\xi}^{1}p^{\downarrow})_{6,4}$$

$$\xrightarrow{Co}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{6,4}$$

$$b(\begin{bmatrix}0\\0\\0\\0\end{bmatrix})=\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\quad \mathcal{Z}_2=\{\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\0\\1\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\1\\1\end{bmatrix}\}$$

$$b(\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix})=\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\quad \mathcal{Z}_3=\{\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\1\\1\end{bmatrix}\}$$

$$b(\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix})=\begin{bmatrix}max(\frac{2}{5},\frac{1}{2})\\\frac{4}{5}\\0\\1\end{bmatrix}\notin p^{\downarrow}$$

$$b(\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix})=\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\quad \mathcal{Z}_3=\{\begin{bmatrix}1\\\frac{4}{5}\\0\\1\end{bmatrix}\}$$

$$b(\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix})=\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\quad \mathcal{Z}_4=\{\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\1\\1\end{bmatrix}\}$$

$$b(\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix})=\begin{bmatrix}max(\frac{2}{5},\frac{1}{2})\\\frac{4}{5}\\0\\1\end{bmatrix}\notin p^{\downarrow}$$

$$b(\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix})=\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\quad \mathcal{Z}_3=\{\begin{bmatrix}1\\\frac{4}{5}\\0\\1\end{bmatrix}\}$$

Figure 7.6: On the left, the execution of $\mathtt{AdjointPDR}^{\downarrow}{}_{\mathtt{hCo01}}$ for the max reachability problem of Example 7.29: in the last state, it returns true since $x_3 = x_4$. On the right, the data explaining the choices of (Conflict) and (Decide). Note that, in the two (Decide) steps, the guard $b(x_{k-1}) \notin Y_k$ holds because of the possibility of choosing the label b in state s_0. This explain why Z is taken as $\mathcal{F}_{\xi}^{1}(p^{\downarrow})$ for the scheduler ξ defined in (7.11).

7.6 AdjointPDR$^{\mathtt{AI}}$

In Section 7.2 we have introduced `AdjointPDR` and in Section 7.4 `AdjointPDR`$^{\downarrow}$: the latter is inspired by the former but takes the positive chain in some lattice L and the negative sequence in $L^{\downarrow}$. In this section, we introduce `AdjointPDR`$^{\mathtt{AI}}$, a third algorithm that generalises both by allowing to manipulate positive and negative sequences in two different lattices. The interest in this generalisation is not just theoretical, but it is also convenient in practice since it allows us in the next section to use an implementation of `AdjointPDR`$^{\mathtt{AI}}$ as a common template for both `AdjointPDR` and `AdjointPDR`$^{\downarrow}$.

The framework for `AdjointPDR`$^{\mathtt{AI}}$ is a diagram

$$\begin{tikzcd} (L, \leq_L) \arrow[loop left, "b"] \arrow[r, "\gamma"] & (C, \leq_C) \arrow[loop right, "\bar{b} \dashv \bar{b}_r"] \end{tikzcd} \tag{7.12}$$

where $(L, \leq_L)$ and $(C, \leq_C)$ are complete lattices, $b\colon L \to L$ is an ω-continuous function, $\bar{b} \dashv \bar{b}_r\colon C \to C$, and $\gamma : L \to C$ is a function satisfying

1. *order-embeddingness*: $x \leq y$ if and only if $\gamma(x) \leq \gamma(y)$ for all $x, y \in L$, and
2. *forward completeness*: $\bar{b}\gamma = \gamma b$.

In this setting the problem $\mu b \leq p$ in L has an equivalent formulation in C:

$$
\begin{aligned}
&(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|\varepsilon)_{3,3} \overset{Ca}{\to} (\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{3,2} \overset{Co}{\to} (\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\|\varepsilon)_{3,3}\\
&\overset{U}{\to}\overset{Ca}{\to}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{4,3} \overset{Co}{\to} (\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\|\varepsilon)_{4,4}\\
&\overset{U}{\to}\overset{Ca}{\to}(\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}1\\1\\1\\1\end{bmatrix}\|p^{\downarrow})_{5,4} \overset{Co}{\to} (\emptyset\begin{bmatrix}0\\0\\0\\0\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}\|p^{\downarrow})_{5,4}
\end{aligned}
$$

$$
\begin{aligned}
b(\begin{bmatrix}0\\0\\0\\0\end{bmatrix}) = \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \quad \mathcal{Z}_2 &= \{\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\0\\1\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\1\\1\end{bmatrix}\}\\
b(\begin{bmatrix}\frac{2}{5}\\0\\0\\1\end{bmatrix}) = \begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix} \quad \mathcal{Z}_3 &= \{\begin{bmatrix}\frac{2}{5}\\1\\0\\1\end{bmatrix},\begin{bmatrix}\frac{2}{5}\\1\\1\\1\end{bmatrix}\}\\
b(\begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix}) = \begin{bmatrix}\frac{2}{5}\\\frac{4}{5}\\0\\1\end{bmatrix} \quad \mathcal{Z}_4 &= \mathcal{Z}_3
\end{aligned}
$$

Figure 7.7: On the left, the execution of $\texttt{AdjointPDR}^{\downarrow}{}_{\texttt{hCoB}}$ for the max reachability problem of Example 7.29: in the last state, it returns true since $x_3 = x_4$. On the right, the data explaining the three choices of (Conflict).

Proposition 7.30. *Consider the framework* (7.12) *and let $p \in L$. Then lfp$(b) \leq p$ is equivalent to lfp$(\bar{b} \vee \gamma\bot) \leq \gamma p$.*

Proof.

$$
\begin{aligned}
\text{lfp}(b) \leq p &\iff \forall n \in \mathbb{N}.\ b^n\bot \leq p\\
&\iff \forall n \in \mathbb{N}.\ \gamma b^n\bot \leq \gamma p && [\text{because } \gamma \text{ is order-embedding}]\\
&\iff \forall n \in \mathbb{N}.\ \bar{b}^n\gamma\bot \leq \gamma p && [\text{by the forward completeness}]\\
&\iff \text{lfp}(\bar{b} \vee \gamma\bot) \leq \gamma p && [\text{since } \bar{b} \dashv \bar{b}_r]
\end{aligned}
$$

□

The reader may have recognised some similarities with the proof of Proposition 7.19. Indeed, as we will show in Example 7.32, the latter result is an instance of Proposition 7.30.

Example 7.31. Consider a Galois insertion with a forward complete abstract interpretation $b\colon L \to L$ to a concrete semantic function $\bar{b}\colon C \to C$. If the function $\bar{b}$ has a right adjoint, then this is an instance of (7.12). ■

Example 7.32. It is easy to check that (7.3) is an instance of (7.12). Hereafter, we illustrate how this can be understood in categorical terms. For a complete lattice L and an ω-continuous function $b\colon L \to L$, we shall see L as a **Bool**-enriched category where **Bool** is the monoidal category with two objects $\bot, \top$, one arrow $\bot \to \top$ and monoidal product given by $\wedge$. Then the left Kan extension $\text{Lan}_y(y \circ b)$ along the Yoneda embedding $y\colon L \to \mathbf{Bool}^{L^{\text{op}}}$ is a left adjoint of the induced functor $b^* \triangleq (-) \circ b^{\text{op}}$.

$$\begin{tikzcd} L \arrow[loop left, "b"] \arrow[r, "y"] & \mathbf{Bool}^{L^{\text{op}}} \arrow[loop right, "\text{Lan}_y(y\circ b)\dashv b^*"] \end{tikzcd}$$

The above diagram is the same as (7.3) under the isomorphism $\mathbf{Bool}^{L^{\text{op}}} \cong L^{\downarrow}$. It is an instance of (7.12) because:

1. y is order-embedding since y is full and faithful and
2. y is forward complete since the unit $b \Rightarrow \text{Lan}_y(y \circ b) \circ y$ is an isomorphism by [Kel82, Proposition 4.23] and full and faithfulness of y.

■

```
                    AdjointPDR^AI $(b, p, \gamma\colon L \to C, \bar{b}, \bar{b}_r)$
<INITIALISATION>
  $(\vec{x} \| \vec{y})_{n,k}$ := $(\bot, \top \| \varepsilon)_{2,2}$
<ITERATION>                                         % $\vec{x}, \vec{y}$ not conclusive
  case $(\vec{x} \| \vec{y})_{n,k}$ of
     $\vec{y} = \varepsilon$ and $x_{n-1} \leq p$ :                        %(Unfold)
       $(\vec{x} \| \vec{y})_{n,k}$ := $(\vec{x}, \top \| \varepsilon)_{n+1,n+1}$
     $\vec{y} = \varepsilon$ and $x_{n-1} \not\leq p$ :                        %(Candidate)
       choose $z \in C$ such that $\gamma x_{n-1} \not\leq z$ and $\gamma p \leq z$;
       $(\vec{x} \| \vec{y})_{n,k}$ := $(\vec{x} \| z)_{n,n-1}$
     $\vec{y} \neq \varepsilon$ and $\bar{b}\gamma x_{k-1} \not\leq y_k$ :                    %(Decide)
       choose $z \in C$ such that $\gamma x_{k-1} \not\leq z$ and $\bar{b}_r(y_k) \leq z$;
       $(\vec{x} \| \vec{y})_{n,k}$ := $(\vec{x} \| z, \vec{y})_{n,k-1}$
     $\vec{y} \neq \varepsilon$ and $\gamma b x_{k-1} \leq y_k$ :                     %(Conflict)
       choose $z \in L$ such that $\gamma z \leq y_k$ and $b(x_{k-1} \wedge z) \leq z$;
       $(\vec{x} \| \vec{y})_{n,k}$ := $(\vec{x} \wedge_k z \| \mathsf{tail}(\vec{y}))_{n,k+1}$
  endcase
<TERMINATION>
  if $\exists j \in [0, n-2] \,.\, x_{j+1} \leq x_j$ then return true   % $\vec{x}$ conclusive
  if $y_0$ is defined then return false      % $\vec{y}$ conclusive
```

Figure 7.8: AdjointPDR$^{\mathtt{AI}}$ algorithm checking $\mu b \leq p$.

The algorithm AdjointPDR$^{\mathtt{AI}}$checking $\mu b \leq p$ in L is illustrated in Figure 7.8. It is an adaptation of AdjointPDR algorithm checking $\mathrm{lfp}(\bar{b} \vee \gamma\bot) \leq \gamma p$ in C (that, by Proposition 7.30, is equivalent to $\mathrm{lfp}(b) \leq p$) that takes positive sequences in L.

More precisely, AdjointPDR$^{\mathtt{AI}}$manipulates pairs $(\vec{x} \| \vec{y})_{n,k}$ of sequences $\vec{x}$ in L and $\vec{y}$ in C while AdjointPDRmanipulates pairs of sequences in C. The pairs in AdjointPDR$^{\mathtt{AI}}$ satisfy invariants (I0), (I1), (I2) and (PN'): $\forall j \in [k, n-1]$, $\gamma x_j \not\leq y_j$. Observe that $(\bot, \gamma\vec{x}) \triangleq (\bot, \gamma x_0, \ldots, \gamma x_{n-1})$ forms a positive chain (Definition 7.1) and $\vec{y}$ forms a negative sequence (Definition 7.2) for $\mathrm{lfp}(\bar{b} \vee \gamma\bot) \leq \gamma p$ in C. The algorithm returns true if $\vec{x}$ is conclusive (i.e., $\exists j \in [0, n-2] \,.\, x_{j+1} \leq x_j$) which implies $(\bot, \gamma\vec{x})$ is also conclusive. It returns false if y_0 is defined, or equivalently, if $\vec{y}$ in the pair $(\bot, \gamma\vec{x} \| \vec{y})$ is conclusive. This equivalence is deduced from (I0) and $\gamma x_0 \not\leq y_0$ by (PN').

The above discussion immediately yields the soundness of AdjointPDR$^{\mathtt{AI}}$. One can also prove the properties of canonical choices, impossibility of loops, and negative termination for AdjointPDR$^{\mathtt{AI}}$ in the same way as for AdjointPDR$^{\downarrow}$, yielding the following result.

Theorem 7.33. *All results in Section 7.3, but Proposition 7.6.3, hold for* AdjointPDR^{AI}.

We conclude this section by illustrating how the algorithm AdjointPDR$^{\mathtt{AI}}$ generalises both AdjointPDR and AdjointPDR$^{\downarrow}$.

Proposition 7.34. AdjointPDR *is an instance of* AdjointPDR^{AI}.

Proof. Consider the setting for AdjointPDRwith (i, f, g, p), i.e. $i, p \in L$ and $f \dashv g\colon L \to L$. Let $L_{i\uparrow}$ be the complete lattice $\{x \in L \mid i \leq x\}$ (with the same order as L) and let $s\colon L_{i\uparrow} \hookrightarrow L$ be the inclusion function. Then AdjointPDR$^{\mathtt{AI}}$with parameters $(f \vee i, p, s\colon L_{i\uparrow} \hookrightarrow L, f, g)$ defines exactly AdjointPDR with parameters (i, f, g, p) and starting state $(\bot, i, \top \| \epsilon)_{3,3}$ (reachable in AdjointPDR applying (Candidate), (Conflict) and (Unfold)). □

Proposition 7.35. $\texttt{AdjointPDR}^{\downarrow}$ *is an instance of* $\texttt{AdjointPDR}^{AI}$.

Proof. Consider the setting for $\texttt{AdjointPDR}^{\downarrow}$, i.e. (b, p) with an ω-continuous function $b\colon L \to L$ and $p \in L$. Then $\texttt{AdjointPDR}^{\texttt{AI}}$ with parameters $(b, p, (-)^{\downarrow}\colon L \to L^{\downarrow}, b^{\downarrow}, b_r^{\downarrow})$ defines exactly $\texttt{AdjointPDR}^{\downarrow}$ with parameters (b, p) under the correspondence between intermediate data $(\vec{x}\|\vec{y})_{n,k}$ in $\texttt{AdjointPDR}^{\texttt{AI}}$ and $(\emptyset, \vec{x}\|\vec{Y})$ in $\texttt{AdjointPDR}^{\downarrow}$ where $Y_{i+1} := y_i$ for each $i \in \{k, \dots, n-1\}$. Please note that $x^{\downarrow} \subseteq Z$ if and only if $x \in Z$ for a lower set Z. □

7.7 Implementation

We first developed, using Haskell and exploiting its abstraction features, a common template of $\texttt{AdjointPDR}^{\texttt{AI}}$ that accommodates both AdjointPDR and $\texttt{AdjointPDR}^{\downarrow}$. It is a program parametrized by two lattices—used for positive chains and negative sequences, respectively—and by a heuristic.

For our experiments, we instantiated the template to $\texttt{AdjointPDR}^{\downarrow}$ for MDPs (letting $L = [0, 1]^S$), with three different heuristics: hCoB and hCo01 from Proposition 7.28; and hCoS introduced below. Besides the template (∼100 lines), we needed ∼140 lines to account for hCoB and hCo01, and additional ∼100 lines to further obtain hCoS. All this indicates a clear benefit of our abstract theory: a general template can itself be coded succinctly; instantiation to concrete problems is easy, too, thanks to an explicitly specified interface of heuristics.

Our implementation accepts MDPs expressed in a symbolic format inspired by Prism models [KNP11], in which states are variable valuations and transitions are described by symbolic functions (they can be segmented with symbolic guards $\{\mathrm{guard}_i\}_i$). We use rational arithmetic (Rational in Haskell) for probabilities to avoid rounding errors.

Heuristics. The three heuristics (hCoB, hCo01, hCoS) use the same choices in (Candidate) and (Decide), as defined in (7.7), but different ones in (Conflict).

The third heuristics hCoS is a *symbolic* variant of hCoB; it relies on our symbolic model format. It uses z_S for z in (Conflict), where $z_S(s) = z_B(s)$ if $r_s \neq 0$ or $\mathcal{Z}_k = \emptyset$. The definition of $z_S(s)$ otherwise is notable: we use a piecewise affine function $(t_i \cdot s + u_i)_i$ for $z_S(s)$, where the affine functions $(t_i \cdot s + u_i)_i$ are guarded by the same guards $\{\mathrm{guard}_i\}_i$ of the MDP's transition function. We let the SMT solver Z3 [MB08] search for the values of the coefficients t_i, u_i, so that z_S satisfies the requirements of (Conflict) (namely $b(x_{k-1})(s) \leq z_S(s) \leq 1$ for each $s \in S$ with $r_s = 0$), together with the condition $b(z_S) \leq z_S$ for faster convergence. If the search is unsuccessful, we give up hCoS and fall back on the hCoB.

As a task common to the three heuristics, we need to calculate $\mathcal{Z}_k = \{d \in \mathcal{G}_k \mid b(x_{k-1}) \leq d\}$ in (Conflict) (see (7.10)). Rather than computing the whole set $\mathcal{G}_k$ of generating points of the linear inequality that defines Y_k, we implemented an ad-hoc algorithm that crucially exploits the condition $b(x_{k-1}) \leq d$ for pruning.

Experiment Settings. We conducted the experiments on Ubuntu 18.04 and AWS t2.xlarge (4 CPUs, 16 GB memory, up to 3.0 GHz Intel Scalable Processor). We used several Markov chain (MC) benchmarks and a couple of MDP ones.

Research Questions. We wish to address the following questions.

RQ1 Does $\texttt{AdjointPDR}^{\downarrow}$ advance the state-of-the-art performance of *PDR* algorithms for probabilistic model checking?

Table 7.1: Experimental results on MC benchmarks. $|S|$ is the number of states, P is the reachability probability (calculated by manual inspection), λ is the threshold in the problem $P \leq_? \lambda$ (shaded if the answer is no). The other columns show the average execution time in seconds; TO is timeout (900 s); MO is out-of-memory. For `AdjointPDR`$^\downarrow$ and LT-PDR we used the `tasty-bench` Haskell package and repeated executions until std. dev. is $< 5\%$ (at least three execs). For PrIC3 and Storm, we made five executions. Storm's execution does not depend on λ: it seems to answer queries of the form $P \leq_? \lambda$ by calculating P. We observed a wrong answer for the entry with (†) (Storm, sp.-num., Haddad-Monmege); see the discussion of RQ2.

Benchmark	$\|S\|$	P	λ	`AdjointPDR`$^\downarrow$			LT-PDR	PrIC3				Storm		
				`hCoB`	`hCo01`	`hCoS`		none	lin.	pol.	hyb.	sp.-num.	sp.-rat.	sp.-sd.
Grid	10^2	0.033	0.3	0.013	0.022	0.659	0.343	1.383	23.301	MO	MO	0.010	0.010	0.010
			0.2	0.013	0.031	0.657	0.519	1.571	26.668	TO	MO			
	10^3	<0.001	0.3	1.156	2.187	5.633	126.441	TO	TO	TO	MO	0.010	0.017	0.011
			0.2	1.146	2.133	5.632	161.667	TO	TO	TO	MO			
BRP	10^3	0.035	0.1	12.909	7.969	55.788	TO	TO	TO	MO	MO	0.012	0.018	0.011
			0.01	1.977	8.111	5.645	21.078	60.738	626.052	524.373	823.082			
			0.005	0.604	2.261	2.709	1.429	12.171	254.000	197.940	318.840			
Zero-Conf	10^2	0.5	0.9	1.217	68.937	0.196	TO	19.765	136.491	0.630	0.468	0.010	0.018	0.011
			0.75	1.223	68.394	0.636	TO	19.782	132.780	0.602	0.467			
			0.52	1.228	60.024	0.739	TO	19.852	136.533	0.608	0.474			
			0.45	<0.001	0.001	0.001	<0.001	0.035	0.043	0.043	0.043			
	10^4	0.5	0.9	MO	TO	7.443	TO	TO	TO	0.602	0.465	0.037	262.193	0.031
			0.75	MO	TO	15.223	TO	TO	TO	0.599	0.470			
			0.52	MO	TO	TO	TO	TO	TO	0.488	0.475			
			0.45	0.108	0.119	0.169	0.016	0.035	0.040	0.040	0.040			
Chain	10^3	0.394	0.9	36.083	TO	0.478	TO	269.801	TO	0.938	0.686	0.010	0.014	0.011
			0.4	35.961	TO	394.955	TO	271.885	TO	0.920	TO			
			0.35	101.351	TO	454.892	435.199	238.613	TO	TO	TO			
			0.3	62.036	463.981	120.557	209.346	124.829	746.595	TO	TO			
Double-Chain	10^3	0.215	0.9	12.122	7.318	TO	TO	TO	TO	1.878	2.053	0.011	0.018	0.010
			0.3	12.120	20.424	TO	TO	TO	TO	1.953	2.058			
			0.216	12.096	19.540	TO	TO	TO	TO	172.170	TO			
			0.15	12.344	16.172	TO	16.963	TO	TO	TO	TO			
Haddad-Monmege	41	0.7	0.9	0.004	0.009	8.528	TO	1.188	31.915	TO	MO	0.011	0.011	1.560
			0.75	0.004	0.011	2.357	TO	1.209	32.143	TO	712.086			
	10^3	0.7	0.9	59.721	61.777	TO	TO	TO	TO	TO	TO	0.013 (†)	0.043	TO
			0.75	60.413	63.050	TO	TO	TO	TO	TO	TO			

Table 7.2: Experimental results on MDP benchmarks. The legend is the same as Table 7.1, except that P is now the maximum reachability probability.

Benchmark	$\lvert S\rvert$	P	λ	AdjointPDR$^{\downarrow}$			Storm		
				hCoB	hCo01	hCoS	sp.-num	sp.-rat.	sp.-sd.
CDrive2	38	0.865	0.9	MO	0.172	TO	0.019	0.019	0.018
			0.75	MO	0.058	TO			
			0.5	0.015	0.029	86.798			
TireWorld	8670	0.233	0.9	MO	3.346	TO	0.070	0.164	0.069
			0.75	MO	3.337	TO			
			0.5	MO	6.928	TO			
			0.2	4.246	24.538	TO			

RQ2 How does AdjointPDR$^{\downarrow}$'s performance compare against *non-PDR* algorithms for probabilistic model checking?

RQ3 Does the theoretical framework of AdjointPDR$^{\downarrow}$ successfully guide the discovery of various heuristics with practical performance?

RQ4 Does AdjointPDR$^{\downarrow}$ successfully manage nondeterminism in MDPs (that is absent in MCs)?

Experiments on MCs (Table 7.1). We used six benchmarks: Haddad-Monmege is from [Har+19]; the others are from [Bat+20; Kor+22]. We compared AdjointPDR$^{\downarrow}$(with three heuristics) against LT-PDR [Kor+22], PrIC3 (with four heuristics *none*, *lin.*, *pol.*, *hyb.*, see [Bat+20]), and Storm 1.5 [DJKV17]. Storm is a recent comprehensive toolsuite that implements different algorithms and solvers. Among them, our comparison is against *sparse-numeric*, *sparse-rational*, and *sparse-sound.* The *sparse* engine uses explicit state space representation by sparse matrices; this is unlike another representative *dd* engine that uses symbolic BDDs. (We did not use *dd* since it often reported errors, and was overall slower than *sparse*.) *Sparse-numeric* is a value-iteration (VI) algorithm; *sparse-rational* solves linear (in)equations using rational arithmetic; *sparse-sound* is a sound VI algorithm [QK18].[1]

Experiments on MDPs (Table 7.2). We used two benchmarks from [Har+19]. We compared AdjointPDR$^{\downarrow}$only against Storm, since RQ1 is already addressed using MCs (besides, PrIC3 did not run for MDPs).

Discussion. The experimental results suggest the following answers to the RQs.

RQ1. The performance advantage of AdjointPDR$^{\downarrow}$, over both LT-PDR and PrIC3, was clearly observed throughout the benchmarks. AdjointPDR$^{\downarrow}$ outperformed LT-PDR, thus confirming empirically the theoretical observation in Section 7.4.2. The profit is particularly evident in those instances whose answer is positive. AdjointPDR$^{\downarrow}$ generally outperformed PrIC3, too. Exceptions are in ZeroConf, Chain and DoubleChain, where PrIC3 with polynomial (pol.) and hybrid (hyb.) heuristics performs well. This seems to be thanks to the expressivity of the polynomial template in PrIC3, which is a possible

[1] There are two more sound algorithms in Storm: one that utilizes interval iteration [Bai+17] and the other does optimistic VI [HK20]. We have excluded them from the results since we observed that they returned incorrect answers.

enhancement we are yet to implement (currently our symbolic heuristic hCoS uses only the affine template).

RQ2. The comparison with Storm is interesting. Note first that Storm's *sparse-numeric* algorithm is a VI algorithm that gives a guaranteed lower bound *without guaranteed convergence*. Therefore its positive answer to $P \leq_{?} \lambda$ may not be correct. Indeed, for Haddad-Monmege with $|S| \sim 10^3$, it answered $P = 0.5$ which is wrong ((†) in Table 7.1). This is in contrast with PDR algorithms that discovers an explicit witness for $P \leq \lambda$ via their positive chain.

Storm's *sparse-rational* algorithm is precise. It was faster than PDR algorithms in many benchmarks, although AdjointPDR$^{\downarrow}$was better or comparable in ZeroConf (10^4) and Haddad-Monmege (41), for λ such that $P \leq \lambda$ is true. We believe this suggests a general advantage of PDR algorithms, namely to accelerate the search for an invariant-like witness for safety.

Storm's *sparse-sound* algorithm is a sound VI algorithm that returns correct answers aside numerical errors. Its performance was similar to that of sparse-numeric, except for the two instances of Haddad-Monmege: sparse-sound returned correct answers but was much slower than sparse-numeric. For these two instances, AdjointPDR$^{\downarrow}$ outperformed sparse-sound.

It seems that a big part of Storm's good performance is attributed to the sparsity of state representation. This is notable in the comparison of the two instances of Haddad-Monmege (41 vs. 10^3): while Storm handles both of them easily, AdjointPDR$^{\downarrow}$ struggles a bit in the bigger instance. Our implementation can be extended to use sparse representation, too; this is future work.

RQ3. We derived the three heuristics (hCoB, hCo01, hCoS) exploiting the theory of AdjointPDR$^{\downarrow}$. The experiments show that each heuristic has its own strength. For example, hCo01 is slower than hCoB for MCs, but it is much better for MDPs. In general, there is no silver bullet heuristic, so coming up with a variety of them is important. The experiments suggest that our theory of AdjointPDR$^{\downarrow}$ provides great help in doing so.

RQ4. Table 7.2 shows that AdjointPDR$^{\downarrow}$ can handle nondeterminism well: once a suitable heuristic is chosen, its performances on MDPs and on MCs of similar size are comparable. It is also interesting that better-performing heuristics vary, as we discussed above.

Summary. AdjointPDR$^{\downarrow}$ clearly outperforms existing probabilistic PDR algorithms in many benchmarks. It also compares well with Storm—a highly sophisticated toolsuite—in a couple of benchmarks. These are notable especially given that AdjointPDR$^{\downarrow}$ currently lacks enhancing features such as richer symbolic templates and sparse representation (adding which is future work). Overall, we believe that AdjointPDR$^{\downarrow}$ *confirms the potential of PDR algorithms in probabilistic model checking.* Through the three heuristics, we also observed the value of an abstract general theory in devising heuristics in PDR, which is probably true of verification algorithms in general besides PDR.

7.8 Conclusions

In this chapter, we presents AdjointPDR, an algorithm that generalizes Bradley's PDR [Bra11] to address the least fixpoint problem $\mathrm{lfp}(f \vee i) \leq p$. The novelty in the algorithm lies in the use of a right adjoint $g \colon L \to L$ of the map f to search for counterexamples: the function f is used in the search of an over-approximation to show that the least fixpoint is below p, while the adjoint g produces candidate counterexamples – in a sense, under-approximations – to witness the violation of the property.

Similar to other PDR-like algorithms, `AdjointPDR` depends on some non-deterministic choices of elements $z \in L$. Therefore, not only we proved soundness of the algorithm for any possible resolution of nondeterminism, but we also showed that the algorithm always progresses to a new state. In general, `AdjointPDR` is not guaranteed to terminate since the problem $\mathrm{lfp}(f \vee i) \leq p$ is generally undecidable. However, we showed that certain well-behaved heuristics, which essentially fix a legitimate choice of z, ensure termination whenever the problem has a negative answer.

The assumption that f possesses a right adjoint g is not satisfied in several interesting cases, particularly those involving probabilistic systems. To address these, we introduce a variation of the algorithm, called $\texttt{AdjointPDR}^{\downarrow}$, that employs lower sets to guarantee the presence of a right adjoint. In this algorithm, a witness for the positive answer is sought in the lattice L, while a counterexample in the lattice of lower sets $L^{\downarrow}$. We demonstrated that most properties of `AdjointPDR` hold for $\texttt{AdjointPDR}^{\downarrow}$ and that $\texttt{AdjointPDR}^{\downarrow}$ simulates LT-PDR [Kor+22], a preceding lattice-theoretical generalization of PDR. Conversely, LT-PDR cannot simulate $\texttt{AdjointPDR}^{\downarrow}$: indeed, the use of $L^{\downarrow}$ enables $\texttt{AdjointPDR}^{\downarrow}$ to simultaneously search for multiple – even all – counterexamples computed by LT-PDR at the same time.

We instantiated $\texttt{AdjointPDR}^{\downarrow}$ to address the max-reachability problem of Markov Decision Processes. Specifically, we devised two heuristics, named `hCoB` and `hCo01`, which cleverly reduce the search space and are guaranteed to terminate in the case of a negative answer. We conducted experimental comparisons of the two heuristics with other tools, yielding promising results: our implementation clearly outperforms other PDR-based algorithms in many benchmarks, and even compares favourably with state-of-the-art tools in a couple of benchmarks. Our implementation is based on a template algorithm, called $\texttt{AdjointPDR}^{\texttt{AI}}$, that generalizes both `AdjointPDR` and $\texttt{AdjointPDR}^{\downarrow}$.

Chapter 8

Conclusions

The main subject of this proposal is combining over- and under-approximation for abstract interpretation based static analysis. The idea of using under-approximation in static analysis is not new, but the focus on exploiting it to *identify true errors* is very recent. For this reason, the subject of under-approximation is largely unexplored yet, and in particular its combination with over-approximation. One of the first task we undertook was to identify past works that used this idea.

Chapter 1 introduces the topic and its motivations. Chapter 2 lays the background needed by subsequent chapters. Chapter 3 discusses some state-of-the-art techniques combining over- and under-approximation, not limited to abstract interpretation based ones. In fact, it examines a common algebraic formalism (Section 3.1), LCL_A (Section 3.2) and `ic3`/PDR (Section 3.3), but only the second employs abstract interpretation.

As a short term goal, we would like to advance these results further, integrating forward repair and domain simplification in LCL_A as well. Moreover, we would like to introduce abstract interpretation in LT-PDR (a generalization of `ic3`), and use it to better understand the trade-offs in the algorithm. In the longer term, we would like to deepen our understanding of over/under-approximation interaction, in order to be able to apply it to more and more techniques. The example of `ic3`, which quickly emerged among the best model checkers, shows that the idea is very powerful. However, it's far from trivial to exploit effectively, hence our research goal.

Bibliography

[Flo67] Robert W. Floyd. "Assigning Meanings to Programs". In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32. URL: http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf.

[Hoa69] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[Law69] F. William Lawvere. "Adjointness in foundations". In: *Dialectica* (1969), pp. 281–296.

[Dij70] Edsger W. Dijkstra. "Notes on Structured Programming". circulated privately. Apr. 1970. URL: http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

[HL74] C. A. R. Hoare and Peter E. Lauer. "Consistent and Complementary Formal Theories of the Semantics of Programming Languages". In: *Acta Informatica* 3 (1974), pp. 135–153. DOI: 10.1007/BF00264034.

[Dij75] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973.

[Coo78] Stephen A. Cook. "Soundness and Completeness of an Axiom System for Program Verification". In: *SIAM J. Comput.* 7.1 (1978), pp. 70–90. DOI: 10.1137/0207005.

[CH78] Patrick Cousot and Nicolas Halbwachs. "Automatic Discovery of Linear Restraints among Variables of a Program". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '78. Tucson, Arizona: Association for Computing Machinery, 1978, pp. 84–96. ISBN: 9781450373487. DOI: 10.1145/512760.512770.

[Hoa78] C. A. R. Hoare. "Some Properties of Predicate Transformers". In: *J. ACM* 25.3 (1978), pp. 461–480. DOI: 10.1145/322077.322088.

[CC79] Patrick Cousot and Radhia Cousot. "Systematic Design of Program Analysis Frameworks". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: Association for Computing Machinery, 1979, pp. 269–282. ISBN: 9781450373579. DOI: 10.1145/567752.567778.

[Apt81] Krzysztof R. Apt. "Ten Years of Hoare's Logic: A Survey—Part I". In: *ACM Trans. Program. Lang. Syst.* 3.4 (Oct. 1981), pp. 431–483. ISSN: 0164-0925. DOI: 10.1145/357146.357150.

[Kel82] Gregory Maxwell Kelly. *Basic concepts of enriched category theory.* Vol. 64. CUP Archive, 1982.

[Apt84] Krzysztof R. Apt. "Ten Years of Hoare's Logic: A Survey Part II: Nondeterminism". In: *Theor. Comput. Sci.* 28 (1984), pp. 83–109. DOI: 10.1016/0304-3975(83)90066-X.

[Mil89] R. Milner. *Communication and Concurrency.* USA: Prentice-Hall, Inc., 1989. ISBN: 0131149849.

[Gra91] Philippe Granger. "Static Analysis of Linear Congruence Equalities among Variables of a Program". In: *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP'91).* Ed. by Samson Abramsky and T. S. E. Maibaum. Vol. 493. Lecture Notes in Computer Science. Springer, 1991, pp. 169–192. DOI: 10.1007/3-540-53982-4_10.

[CC92] Patrick Cousot and Radhia Cousot. "Abstract Interpretation Frameworks". In: *J. Log. Comput.* 2.4 (1992), pp. 511–547. DOI: 10.1093/LOGCOM/2.4.511.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: an Introduction.* MIT press, 1993.

[GS97] Susanne Graf and Hassen Saïdi. "Construction of Abstract State Graphs with PVS". In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings.* Ed. by Orna Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer, 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10.

[Koz97] Dexter Kozen. "Kleene Algebra with Tests". In: *ACM Trans. Program. Lang. Syst.* 19.3 (May 1997), pp. 427–443. ISSN: 0164-0925. DOI: 10.1145/256167.256195.

[Cla+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-Guided Abstraction Refinement". In: *Proc. of CAV'00.* Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Springer, 2000, pp. 154–169. ISBN: 978-3-540-45047-4. DOI: 10.1007/10722167_15.

[GRS00] Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. "Making Abstract Interpretations Complete". In: *J. ACM* 47.2 (Mar. 2000), pp. 361–416. ISSN: 0004-5411. DOI: 10.1145/333979.333989.

[Koz00] Dexter Kozen. "On Hoare Logic and Kleene Algebra with Tests". In: *ACM Trans. Comput. Logic* 1.1 (July 2000), pp. 60–76. ISSN: 1529-3785. DOI: 10.1145/343369.343378.

[GQ01] Roberto Giacobazzi and Elisa Quintarelli. "Incompleteness, Counterexamples, and Refinements in Abstract Model-Checking". In: *Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings.* Ed. by Patrick Cousot. Vol. 2126. Lecture Notes in Computer Science. Springer, 2001, pp. 356–373. DOI: 10.1007/3-540-47764-0_20.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local Reasoning about Programs that Alter Data Structures". In: *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings.* Ed. by Laurent Fribourg. Vol. 2142. Lecture Notes in Computer Science. Springer, 2001, pp. 1–19. DOI: `10.1007/3-540-44802-0\_1`.

[DP02] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* 2nd ed. Cambridge University Press, 2002. ISBN: 9780511809088. DOI: `10.1017/CBO9780511809088`.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Lazy abstraction". In: *Proc. of POPL'02.* Ed. by John Launchbury and John C. Mitchell. ACM, 2002, pp. 58–70. DOI: `10.1145/503272.503279`.

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 2002, pp. 55–74. DOI: `10.1109/LICS.2002.1029817`.

[Lev04] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis.* Vol. 2. Semantics Structures in Computation. Springer, 2004. ISBN: 1-4020-1730-8.

[BC05] Marc Bezem and Thierry Coquand. "Automating Coherent Logic". In: *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings.* Ed. by Geoff Sutcliffe and Andrei Voronkov. Vol. 3835. Lecture Notes in Computer Science. Springer, 2005, pp. 246–260. DOI: `10.1007/11591191\_18`.

[DMS06] Jules Desharnais, Bernhard Möller, and Georg Struth. "Kleene Algebra with Domain". In: *ACM Trans. Comput. Logic* 7.4 (Oct. 2006), pp. 798–833. ISSN: 1529-3785. DOI: `10.1145/1183278.1183285`.

[McM06] Kenneth L. McMillan. "Lazy Abstraction with Interpolants". In: *Proc. of CAV'06.* Ed. by Thomas Ball and Robert B. Jones. Vol. 4144. LNCS. Springer, 2006, pp. 123–136. DOI: `10.1007/11817963\_14`.

[Min06] A. Miné. "The octagon abstract domain". In: *High. Order Symb. Comput.* 19.1 (2006), pp. 31–100. DOI: `10.1007/s10990-006-8609-1`.

[LSRG07] Tal Lev-Ami, Mooly Sagiv, Thomas Reps, and Sumit Gulwani. "Backward analysis for inferring quantified preconditions". In: *Tr-2007-12-01, Tel Aviv University* (2007).

[Sch07] David A. Schmidt. "A calculus of logical relations for over- and underapproximating static analyses". In: *Sci. Comput. Program.* 64.1 (2007), pp. 29–53. DOI: `10.1016/j.scico.2006.03.008`.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT Press, 2008. ISBN: 978-0-262-02649-9.

[MB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An Efficient SMT Solver". In: *Proc. of TACAS 2008.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: `10.1007/978-3-540-78800-3_24`.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. "Compositional shape analysis by means of bi-abduction". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 289–300. DOI: 10.1145/1480881.1480917.

[LL09] Vincent Laviron and Francesco Logozzo. "Refining Abstract Interpretation-Based Static Analyses with Hints". In: *Proc. of APLAS'09*. Ed. by Zhenjiang Hu. Vol. 5904. LNCS. Springer, 2009, pp. 343–358. DOI: 10.1007/978-3-642-10672-9_24.

[Bra11] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Ed. by Ranjit Jhala and David A. Schmidt. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. "Precondition Inference from Intermittent Assertions and Application to Contracts on Collections". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Ed. by Ranjit Jhala and David A. Schmidt. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 150–168. DOI: 10.1007/978-3-642-18275-4_12.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. "PRISM 4.0: Verification of Probabilistic Real-Time Systems". In: *Proc. of CAV 2011*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[VK11] Edsko de Vries and Vasileios Koutavas. "Reverse Hoare Logic". In: *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Vol. 7041. Lecture Notes in Computer Science. Springer, 2011, pp. 155–171. DOI: 10.1007/978-3-642-24690-6_12.

[Bra12] Aaron R. Bradley. "Understanding IC3". In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. Lecture Notes in Computer Science. Springer, 2012, pp. 1–14. DOI: 10.1007/978-3-642-31612-8_1.

[CG12] Alessandro Cimatti and Alberto Griggio. "Software Model Checking via IC3". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 277–293. DOI: 10.1007/978-3-642-31424-7_23.

[CCLB12] Patrick Cousot, Radhia Cousot, Francesco Logozzo, and Michael Barnett. "An Abstract Interpretation Framework for Refactoring with Application to Extract Methods with Contracts". In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 213–232. DOI: 10.1145/2384616.2384633.

[CCFL13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. "Automatic Inference of Necessary Preconditions". In: *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*. Ed. by Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni. Vol. 7737. Lecture Notes in Computer Science. Springer, 2013, pp. 128–148. DOI: 10.1007/978-3-642-35873-9_10.

[CC14] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: Past, Present and Future". In: *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. CSL-LICS '14. Vienna, Austria: Association for Computing Machinery, 2014. ISBN: 9781450328869. DOI: 10.1145/2603088.2603165.

[Min14] Antoine Miné. "Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions". In: *Sci. Comput. Program.* 93 (Nov. 2014), pp. 154–182. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.09.014.

[GLR15] 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.

[SPV15] Gagandeep Singh, Markus Püschel, and Martin Vechev. "Making Numerical Program Analysis Fast". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 303–313. ISBN: 9781450334686. DOI: 10.1145/2737924.2738000.

[Ass+17] Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. "Hypercollecting Semantics and Its Application to Static Analysis of Information Flow". In: *SIGPLAN Not.* 52.1 (Jan. 2017), pp. 874–887. ISSN: 0362-1340. DOI: 10.1145/3093333.3009889.

[Bai+17] Christel Baier, Joachim Klein, Linda Leuschner, David Parker, and Sascha Wunderlich. "Ensuring the Reliability of Your Model Checker: Interval Iteration for Markov Decision Processes". In: *Proc. of CAV 2017, Part I*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10426. Lecture Notes in Computer Science. Springer, 2017, pp. 160–180. DOI: 10.1007/978-3-319-63387-9_8.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. "A Storm is Coming: A Modern Probabilistic Model Checker". In: *Proc. of CAV 2017, Part II*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427.

Lecture Notes in Computer Science. Springer, 2017, pp. 592–600. DOI: 10.1007/978-3-319-63390-9_31.

[SS17] Tobias Seufert and Christoph Scholl. "Sequential Verification Using Reverse PDR". In: *Proc. of MBMV 2017.* Ed. by Daniel Große and Rolf Drechsler. Shaker Verlag, 2017, pp. 79–90.

[SPV17a] Gagandeep Singh, Markus Püschel, and Martin Vechev. "A Practical Construction for Decomposing Numerical Abstract Domains". In: 2.POPL (Dec. 2017). DOI: 10.1145/3158143.

[SPV17b] Gagandeep Singh, Markus Püschel, and Martin Vechev. "Fast Polyhedra Abstract Domain". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* POPL 2017. Paris, France: Association for Computing Machinery, 2017, pp. 46–59. ISBN: 9781450346603. DOI: 10.1145/3009837.3009885.

[BGGP18] Filippo Bonchi, Pierre Ganty, Roberto Giacobazzi, and Dusko Pavlovic. "Sound up-to techniques and Complete abstract domains". In: *Proc. of LICS 2018.* Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 175–184. DOI: 10.1145/3209108.3209169.

[QK18] Tim Quatmann and Joost-Pieter Katoen. "Sound Value Iteration". In: *Proc. of CAV 2018, Part I.* Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10981. Lecture Notes in Computer Science. Springer, 2018, pp. 643–661. DOI: 10.1007/978-3-319-96145-3_37.

[AO19] Krzysztof R. Apt and Ernst-Rüdiger Olderog. "Fifty years of Hoare's logic". In: *Formal Aspects Comput.* 31.6 (2019), pp. 751–807. DOI: 10.1007/S00165-019-00501-3.

[BC19] Graham Bleaney and Sinan Cepel. *Pysa: An Open Source Static Analysis Tool to Detect and Prevent Security Issues in Python Code.* https://engineering.fb.com/2021/09/29/security/mariana-trench/. 2019.

[Bru+19] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. "Abstract Extensionality: On the Properties of Incomplete Abstract Interpretations". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371096.

[DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. "Scaling Static Analyses at Facebook". In: *Commun. ACM* 62.8 (2019), pp. 62–70. DOI: 10.1145/3338112.

[Har+19] Arnd Hartmanns, Michaela Klauck, David Parker, Tim Quatmann, and Enno Ruijters. "The Quantitative Verification Benchmark Set". In: *Proc. of TACAS 2019, Part I.* Ed. by Tomás Vojnar and Lijun Zhang. Vol. 11427. Lecture Notes in Computer Science. Springer, 2019, pp. 344–350. DOI: 10.1007/978-3-030-17462-0_20.

[Bat+20] Kevin Batz, Sebastian Junges, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Philipp Schröer. "PrIC3: Property Directed Reachability for MDPs". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 512–538. DOI: 10.

1007/978-3-030-53291-8_27. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_27.

[HK20] Arnd Hartmanns and Benjamin Lucien Kaminski. "Optimistic Value Iteration". In: *Proc. of CAV 2020, Part II.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 488–511. DOI: 10.1007/978-3-030-53291-8_26.

[LNNK20] Tim Lange, Martin R. Neuhäußer, Thomas Noll, and Joost-Pieter Katoen. "IC3 software model checking". In: *Int. J. Softw. Tools Technol. Transf.* 22.2 (2020), pp. 135–161. DOI: 10.1007/s10009-019-00547-x.

[OHe20] Peter W. O'Hearn. "Incorrectness logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 10:1–10:32. DOI: 10.1145/3371078.

[Raa+20] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. "Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 225–252. DOI: 10.1007/978-3-030-53291-8_14.

[RY20] X. Rival and K. Yi. *Introduction to Static Analysis – An Abstract Interpretation Perspective.* MIT Press, Feb. 2020. ISBN: 9780262356657.

[BGGR21] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. "A Logic for Locally Complete Abstract Interpretations". In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021.* IEEE, 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470608.

[Gab21] Dominik Gabi. *Open-sourcing Mariana Trench: Analyzing Android and Java App Security in Depth.* https://engineering.fb.com/2021/09/29/security/mariana-trench/. 2021.

[MOH21] Bernhard Möller, Peter W. O'Hearn, and Tony Hoare. "On Algebra of Program Correctness and Incorrectness". In: *Relational and Algebraic Methods in Computer Science - 19th International Conference, RAMiCS 2021, Marseille, France, November 2-5, 2021, Proceedings.* Ed. by Uli Fahrenberg, Mai Gehrke, Luigi Santocanale, and Michael Winter. Vol. 13027. Lecture Notes in Computer Science. Springer, 2021, pp. 325–343. DOI: 10.1007/978-3-030-88701-8_20.

[ABG22] Flavio Ascari, Roberto Bruni, and Roberta Gori. "Limits and difficulties in the design of under-approximation abstract domains". In: *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings.* Ed. by Patricia Bouyer and Lutz Schröder. Vol. 13242. Lecture Notes in Computer Science. Springer, 2022, pp. 21–39. DOI: 10.1007/978-3-030-99253-8_2.

[BRZ22] Paolo Baldan, Francesco Ranzato, and Linpeng Zhang. "Intensional Kleene and Rice theorems for abstract program semantics". In: *Inf. Comput.* 289.Part (2022), p. 104953. DOI: 10.1016/J.IC.2022.104953.

[BGGR22] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. “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.

[Kor+22] Mayuko Kori, Natsuki Urabe, Shin-ya Katsumata, Kohei Suenaga, and Ichiro Hasuo. “The Lattice-Theoretic Essence of Property Directed Reachability Analysis”. In: *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I.* Ed. by Sharon Shoham and Yakir Vizel. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 235–256. DOI: 10.1007/978-3-031-13185-1_12.

[Le+22] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. “Finding Real Bugs in Big Programs with Incorrectness Logic”. In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–27. DOI: 10.1145/3527325.

[MR22] Marco Milanese and Francesco Ranzato. “Local Completeness Logic on Kleene Algebra with Tests”. In: *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings.* Ed. by Gagandeep Singh and Caterina Urban. Vol. 13790. Lecture Notes in Computer Science. Springer, 2022, pp. 350–371. DOI: 10.1007/978-3-031-22308-2_16.

[RBDO22] Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. “Concurrent Incorrectness Separation Logic”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–29. DOI: 10.1145/3498695.

[ZAG22] Cheng Zhang, Arthur Azevedo de Amorim, and Marco Gaboardi. “On Incorrectness Logic and Kleene Algebra with Top and Tests”. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022). DOI: 10.1145/3498690.

[ZK22] Linpeng Zhang and Benjamin Lucien Kaminski. “Quantitative strongest post: a calculus for reasoning about the flow of quantitative information”. In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–29. DOI: 10.1145/3527331.

[ABG23] Flavio Ascari, Roberto Bruni, and Roberta Gori. “Logics for Extensional, Locally Complete Analysis via Domain Refinements”. In: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings.* Ed. by Thomas Wies. Vol. 13990. Lecture Notes in Computer Science. Springer, 2023, pp. 1–27. DOI: 10.1007/978-3-031-30044-8_1.

[BGGR23] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. “A Correctness and Incorrectness Program Logic”. In: *J. ACM* 70.2 (2023), 15:1–15:45. DOI: 10.1145/3582267.

[Kor+23] Mayuko Kori, Flavio Ascari, Filippo Bonchi, Roberto Bruni, Roberta Gori, and Ichiro Hasuo. “Exploiting Adjoints in Property Directed Reachability Analysis”. In: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II.* Ed. by Constantin Enea and Akash Lal. Vol. 13965. Lecture Notes in Computer Science. Springer, 2023, pp. 41–63. DOI: 10.1007/978-3-031-37703-7_3.

[RVBO23] Azalea Raad, Julien Vanegue, Josh Berdine, and Peter W. O'Hearn. "A General Approach to Under-Approximate Reasoning About Concurrent Programs". In: *34th International Conference on Concurrency Theory, CONCUR 2023, September 18-23, 2023, Antwerp, Belgium*. Ed. by Guillermo A. Pérez and Jean-François Raskin. Vol. 279. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 25:1–25:17. DOI: `10.4230/LIPICS.CONCUR.2023.25`.

[ZDS23] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. "Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 522–550. DOI: `10.1145/3586045`.

[ABG24] Flavio Ascari, Roberto Bruni, and Roberta Gori. "Limits and Difficulties in the Design of Under-Approximation Abstract Domains". In: *ACM Trans. Program. Lang. Syst.* (June 2024). Just Accepted. ISSN: 0164-0925. DOI: `10.1145/3666014`.

[ABGL24] Flavio Ascari, Roberto Bruni, Roberta Gori, and Francesco Logozzo. *Sufficient Incorrectness Logic: SIL and Separation SIL*. 2024. arXiv: `2310.18156 [cs.LO]`.

[RVO24] Azalea Raad, Julien Vanegue, and Peter O'Hearn. *Compositional Non-Termination Proving*. Preprint. 2024. URL: `https://www.soundandcomplete.org/papers/Unter.pdf`.

[Zil24] Noam Zilberstein. *A Relatively Complete Program Logic for Effectful Branching*. Preprint. 2024. arXiv: `2401.04594 [cs.LO]`.

[ZSS24] Noam Zilberstein, Angelina Saliling, and Alexandra Silva. "Outcome Separation Logic: Local Reasoning for Correctness and Incorrectness with Computational Effects". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (2024), pp. 276–304. DOI: `10.1145/3649821`. URL: `https://doi.org/10.1145/3649821`.

Appendix A

Appendix

Check that all results have a proof

This appendix contains technical details of proofs and examples for Chapters 2 and 3.

Proof of Proposition 2.4. The proof is by induction on the structure of r.

Case e

The thesis is exactly the hypothesis that $(\!|\cdot|\!)$ is monotone (resp. additive).

Case $\mathsf{r}_1 ; \mathsf{r}_2$

By inductive hypothesis, both $\llbracket \mathsf{r}_1 \rrbracket$ and $\llbracket \mathsf{r}_2 \rrbracket$ are monotone (resp. additive). The thesis follows since composition of monotone (resp. additive) functions is monotone (resp. additive).

Case $\mathsf{r}_1 \oplus \mathsf{r}_2$

By inductive hypothesis, both $\llbracket \mathsf{r}_1 \rrbracket$ and $\llbracket \mathsf{r}_2 \rrbracket$ are monotone (resp. additive). The thesis follows since join of monotone (resp. additive) functions is monotone (resp. additive).

Case $\mathsf{r}^\star$

By inductive hypothesis, $\llbracket \mathsf{r} \rrbracket$ is monotone (resp. additive). Since composition of monotone (resp. additive) functions is monotone (resp. additive), also $\llbracket \mathsf{r} \rrbracket^n$ is monotone (resp. additive). Therefore, $\llbracket \mathsf{r}^\star \rrbracket$ is monotone (resp. additive) because it's a lub of monotone (resp. additive) functions. □

Appendix B

Under-approximation abstract domains supplementary materials

This appendix contains technical details of proofs and examples for Chapter 4.

Proof of Lemma 4.7. By hypothesis c is representable but the pair $\{c, \tilde{c}\}$ is not representable. Since α is monotone and c is representable we have $\alpha(\{c, \tilde{c}\}) \supseteq \alpha(\{c\}) = \{c\}$. Since by correctness $\{c, \tilde{c}\} \supseteq \alpha(\{c, \tilde{c}\})$ and $\alpha(\{c, \tilde{c}\}) \neq \{c, \tilde{c}\}$ because this pair is not representable and hence not in the image of α, it must be the case that $\alpha(\{c, \tilde{c}\}) = \{c\}$.

Now

$$\alpha(f(\{c, \tilde{c}\})) = \alpha(\{f(c), f(\tilde{c})\}) \supseteq \alpha(\{f(\tilde{c})\}) = \{f(\tilde{c})\}$$

where the last equality follows by the hypothesis that $f(\tilde{c}) \in R$. This in particular means that $\alpha(f(\{c, \bar{c}\})) \neq \emptyset$, and together with the fact that $\alpha(\{c, \bar{c}\}) = \{c\} \neq \emptyset$, we get that $f^A(\alpha(\{c, \bar{c}\})) \neq \emptyset$, because f is non-emptying.

From this it follows:

$$\begin{aligned}
\emptyset &\subset f^A(\alpha(\{c, \tilde{c}\})) && \text{[shown above]}\\
&= f^A(\{c\}) && [\alpha(\{c, \tilde{c}\}) = \{c\}]\\
&= \alpha(f(\{c\})) && \text{[definition of } f^A]\\
&= \alpha(\{f(c)\}) && \text{[additivity of } f]\\
&\subseteq \{f(c)\} && \text{[correctness]}
\end{aligned}$$

Since $\alpha(\{f(c)\})$ cannot be empty it must be exactly $\alpha(\{f(c)\}) = \{f(c)\}$, that is $f(c) \in R$. □

Proof of Lemma 4.9. By union closure of the abstract domain, any set $S \cup T$ for $T \subseteq R(S)$ is representable too, since it can be expressed as the union of representable sets:

$$S \cup T = \bigcup_{x \in T} (S \cup \{x\})$$

and each $S \cup \{x\}$ is representable because $x \in T \subseteq R(S)$. The number of those sets is given by the cardinality of $\mathcal{P}(R(S))$. If $R(S)$ were infinite, it would be at least countable, so its powerset $\mathcal{P}(R(S))$ would have a greater cardinality. However, this would conflict with the Assumption 4.8 saying that A is at most countable. Therefore, $R(S)$ must be finite. □

Proof of Lemma 4.12. For any $N \geq N_0$, as in the proof of Lemma 4.9, by union closure any set $S \cup T$ for $T \subseteq R_N(S)$ is representable in A_N. Hence we have

$$\text{poly}(N) = |A_N| \geq |\mathcal{P}(R_N(S))| = 2^{|R_N(S)|}$$

so, taking log at both sides, $|R_N(S)| = O(\log(N))$. □

Proof of Theorem 4.15. Assume by contradiction that A is non-emptying for all $f \in F$. By hypothesis, R is not empty and thus we can take a $c_0 \in R$. We then define recursively a sequence of representable elements c_n.

Given an element $c \in C$, let

$$NR(c) = C \setminus R(c) = \{\tilde{c} \in C \mid \{c, \tilde{c}\} \text{ is not representable}\}$$

be the set of elements that are *not* representable with c. To ease presentation, we define here E_n, a set of elements of C that depends on the sequence c_n that we construct along the proof. Note that the definition of E_n only depends on elements of the sequence up to c_n.

$$\begin{aligned} E_n = \{\tilde{c} \in C \mid & \forall\, 0 \leq i \leq n \;.\; \tilde{c} \in NR(c_i), \\ & f_{\tilde{c}}(\tilde{c}) = c_0, \\ & \forall\, 0 \leq i \leq n-1 \;.\; f_{\tilde{c}}(c_i) = c_{i+1}\} \end{aligned}$$

To improve readability, the conditions on $\tilde{c}$ are separated on three different lines. The first line is needed to make Lemma 4.7 applicable to the pair $\{c_i, \tilde{c}\}$ and conclude that $f_{\tilde{c}}(c_i) = c_{i+1}$ is representable. The second line means just that $\tilde{c}$ represents $f_{\tilde{c}}$, and the last line expresses the requirement that $f_{\tilde{c}}$ coincides on the prefix of the sequence up to c_n.

We observe that the sequence E_n can also be defined inductively by

$$\begin{aligned} E_0 &= \{\tilde{c} \in C \mid \tilde{c} \in NR(c_0), f_{\tilde{c}}(\tilde{c}) = c_0\} \\ &= NR(c_0) \cap P_F(c_0) \end{aligned}$$

$$\begin{aligned} E_{n+1} &= \{\, \tilde{c} \in E_n \mid \tilde{c} \in NR(c_{n+1}), f_{\tilde{c}}(c_n) = c_{n+1}\} \\ &= NR(c_{n+1}) \cap \{\, \tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\} \end{aligned} \tag{B.1}$$

E_0 is the intersection of $NR(c_0)$ and the set of $\tilde{c}$ for which there exists $f_{\tilde{c}}$. Using Lemma 4.9 to say $R(c_0)$ is finite and recalling that $P_F(c_0)$ is infinite by high surjectivity, we observe that

$$E_0 = P_F(c_0) \cap NR(c_0) = P_F(c_0) \setminus (C \setminus NR(c_0)) = P_F(c_0) \setminus R(c_0)$$

is infinite too.

We then prove by induction on n the following three statements:

1. c_n is representable, i.e., $c_n \in R$;

2. E_n is infinite;

3. c_n is different from all c_i for $0 \leq i \leq n-1$.

We have already proved the base case for $n = 0$: c_0 is representable by hypothesis, E_0 is infinite as shown above and the third condition is vacuous since there are no $0 \leq i \leq -1$.

For the inductive step, assume the three hypothesis hold for n and let us prove them for $n+1$. Consider the set

$$S = \{f_{\tilde{c}}(c_n) \mid \tilde{c} \in E_n\}$$

of possible candidates for the role of c_{n+1}. Since $c_n \in R$, $\tilde{c} \in E_n \subseteq NR(c_n)$ and $f_{\tilde{c}}(\tilde{c}) = c_0 \in R$ (by inductive hypotheses) we can apply Lemma 4.7 to get $f_{\tilde{c}}(c_n) \in R$ too, hence S is a subset of R. Since R is finite also S must be, and by inductive hypothesis we know E_n is infinite, so there must be an element c_{n+1} in S such that an infinite amount of $\tilde{c} \in E_n$ satisfies $f_{\tilde{c}}(c_n) = c_{n+1}$. We observe that, as shown above, $c_{n+1} \in R$. Moreover we chose c_{n+1} such that

$$\{\ \tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\}$$

is infinite, so we get

$$\{\ \tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\} \cap NR(c_{n+1}) = \{\ \tilde{c} \in E_n \mid f_{\tilde{c}}(c_n) = c_{n+1}\} \setminus R(c_{n+1})$$

is infinite too because $R(c_{n+1})$ is finite. But this, by Equation (B.1) above, is exactly E_{n+1}.

We only have to show that $c_{n+1} \neq c_i$ for all $0 \leq i \leq n$. Assume by contradiction that this is not the case, so that for some $0 \leq j \leq n$ it holds $c_{n+1} = c_j$. If $f_{\tilde{c}}$ is acyclic this is a contradiction because it would create the cycle $f_{\tilde{c}}^{n+2-j}(c_j) = f_{\tilde{c}}(c_{n+1}) = c_j$. If $f_{\tilde{c}}$ is injective, let us distinguish two cases. If $j = 0$ we get $f_{\tilde{c}}(c_n) = c_{n+1} = c_0 = f_{\tilde{c}}(\tilde{c})$, that would imply $c_n = \tilde{c}$: this is not possible, because the former is representable and the latter is not. If otherwise $j > 0$ we get $f_{\tilde{c}}(c_n) = c_{n+1} = c_j = f_{\tilde{c}}(c_{j-1})$, that would imply $c_n = c_{j-1}$, that is not the case by inductive hypothesis. So $c_{n+1} \neq c_j$, and this concludes the inductive proof.

By the above induction we conclude that all the infinitely many c_n are elements of R and are all distinct. This yields the desired contradiction because R is finite by Lemma 4.9. □

Proof of Theorem 4.19. Towards a contradiction, let us assume that A is non-emptying for all $f \in F$. By hypothesis, R is not empty and thus we can take a $c_0 \in R$.

Since F is a highly surjective family, $P_F(c_0)$ is infinite. By Lemma 4.9 we know that $R(c_0)$ is finite. Therefore we get that the set

$$\begin{aligned} E &= \{\tilde{c} \in C \mid \tilde{c} \notin R(c_0), \exists f_{\tilde{c}} \in F\ .\ f_{\tilde{c}}(\tilde{c}) = c_0\} \\ &= P_F(c_0) \setminus R(c_0) \end{aligned}$$

is infinite. By Lemma 4.7, for all $\tilde{c} \in E$ we have $f_{\tilde{c}}(c_0)$ is representable.

Now fix a function $f \in F$, and let $J(f)$ be the set of $\tilde{c}$ for which f can play the role of $f_{\tilde{c}}$:

$$J(f) = \{\tilde{c} \in C \setminus R(c_0) \mid f(\tilde{c}) = c_0\}$$

By hypothesis (2), the set $J(f)$ is finite.

Now let G be the set of functions in F that can play the role of $f_{\tilde{c}}$ for some $\tilde{c} \in E$:

$$G = \{f \in F \mid \exists \tilde{c} \in E.\ f(\tilde{c}) = c_0\}$$

Clearly

$$E = \bigcup_{f \in G} J(f)$$

Since E is infinite while each $J(f)$ is finite, the set G must be infinite too.

The above observation about $f_{\tilde{c}}(c_0)$ being representable can be equivalently restated by saying that for all $f \in G$, $f(c_0)$ is representable. So consider the set I of all possible images of c_0 through functions in G:

$$I = \{f(c_0) \mid f \in G\}$$

This set is a subset of R because all its elements are representable.

Clearly

$$G = \bigcup_{d \in I} \{f \in G \mid f(c_0) = d\}$$

Now observe that for any $d \in C$, by hypothesis (1) the set

$$\{f \in F \mid f(c_0) = d\}$$

is finite, and this is a superset of $\{f \in G \mid f(c_0) = d\}$, which must be finite too. Since we know that G is infinite, the set I must be infinite too.

This leads to the desired contradiction: R is finite by Lemma 4.9 but we found that I is an infinite set of representable elements. □

Proof of Proposition 4.22. Since F is not highly surjective, there exists $c_0 \in C$ such that $P_F(c_0)$ is finite. We then define A_F as follows. The only element of C representable on its own is c_0 itself, ie. $R = \{c_0\}$. A pair of elements of C is representable if and only if one of its elements is c_0 and the other is in $P_F(c_0)$. This also means that $R(c_0) = P_F(c_0)$. Subsets of C with at least three elements are representable if and only if they are unions of representable pairs. The complete definition of A_F is then

$$A_F = \{\emptyset\} \cup \{\{c_0\} \cup T \mid T \subseteq P_F(c_0)\}$$

A_F is an opposite Moore family with respect to $\mathcal{P}(C)$: it contains the minimal element, that is $\emptyset$, and is closed by union, that are lubs. Hence A_F is an under-approximation abstract domain. Moreover, since $R(c_0) = P_F(c_0)$ is finite, we get that A_F is finite too:

$$|A_F| = 1 + 2^{|P_F(c_0)|}.$$

Now we want to show that an arbitrary f in F is non-emptying in A_F. We first observe that a subset $S \subseteq C$ is such that $\alpha(S) \neq \emptyset$ if and only if $c_0 \in S$. Suppose that $c_0 \in S$, then

$$\alpha(S) \supseteq \alpha(\{c_0\}) = \{c_0\} \supset \emptyset.$$

Conversely, suppose that $\alpha(S) \neq \emptyset$. Since all elements of A_F but the empty set contains c_0, by correctness we have

$$c_0 \in \alpha(S) \subseteq S.$$

So fix now $S \subseteq C$ an element of the concrete domain, and assume that both $\alpha(S) \neq \emptyset$ and $\alpha(f(S)) \neq \emptyset$. These two assumptions are equivalent to the conditions $c_0 \in S$ and $c_0 \in f(S)$, respectively, and the second can be rewritten as $\exists d \in S \ . \ f(d) = c_0$. By definition of A_F we know this is equivalent to $d \in P_F(c_0) = R(c_0)$. Hence

$$\begin{aligned}
& S \supseteq \{c_0, d\} & \\
\Longrightarrow\ & \alpha(S) \supseteq \alpha(\{c_0, d\}) = \{c_0, d\} & [d \in R(c_0)] \\
\Longrightarrow\ & f(\alpha(S)) \supseteq f(\{c_0, d\}) \supseteq \{f(d)\} = \{c_0\} & [f(d) = c_0] \\
\Longrightarrow\ & f^A(\alpha(S)) = \alpha(f(\alpha(S))) \supseteq \alpha(\{c_0\}) = \{c_0\} & [c_0 \in R]
\end{aligned}$$

where in the second line $d \in R(c_0)$ entails $\alpha(\{c_0, d\}) = \{c_0, d\}$. The last line implies $f^A(\alpha(S)) \neq \emptyset$, so f is non-emptying in A_F. □

Proof of Proposition 4.24. Since the proof is very similar to that of Proposition 4.22 above, we gloss over some details.

First, we define a *basis* for the abstract domain as

$$B_F = \{S_0\} \cup \{S_0 \cup S \mid S \in P_F(S_0)\}$$

and then consider its closure under union

$$A_F = \left\{ \bigcup_{T \in \Gamma} T \mid \Gamma \subseteq B_F \right\}.$$

This is an opposite Moore family and hence is an under-approximation abstract domain for $\mathcal{P}(C)$, and is finite because

$$|A_F| \le |\mathcal{P}(B_F)|$$

and

$$|B_F| \le 1 + |\mathcal{P}(P_F(S_0))|$$

with $P_F(S_0)$ finite by hypothesis.

Again we observe that a subset $T \subseteq C$ is such that $\alpha(T) \neq \emptyset$ if and only if $S_0 \subseteq T$ because all elements of the abstract domain but the empty set contains S_0. Then the proof proceeds as above: fix $f \in F$ and $T \subseteq C$ such that $\alpha(T) \neq \emptyset$ and $\alpha(f(T)) \neq \emptyset$, that in turn are equivalent to $S_0 \subseteq T$ and $\exists S \subseteq T. f(S) = S_0$. Then by definition of A_F this means $S_0 \cup S \in A_F$, so

$$\begin{aligned}
& T \supseteq S_0 \cup S && \\
\implies & \alpha(T) \supseteq \alpha(S_0 \cup S) = S_0 \cup S && [S_0 \cup S \in A_F] \\
\implies & f(\alpha(T)) \supseteq f(S_0 \cup S) \supseteq f(S) = S_0 && [f(S) = S_0] \\
\implies & f^A(\alpha(T)) = \alpha(f(\alpha(T))) \supseteq \alpha(S_0) = S_0 && [S_0 \in A_F]
\end{aligned}$$

that is f is non-emptying. □

Proof of Proposition 4.26. Define a *basis* for the abstract domain

$$B_F = \{S_0\} \cup \{S_0 \cup S \mid S \in I_F(S_0)\}$$

and consider its closure under union

$$A_F = \left\{ \bigcup_{T \in \Gamma} T \mid \Gamma \subseteq B_F \right\}.$$

Again this is a correct finite under-approximation abstract domain for $\mathcal{P}(C)$ satisfying that $\alpha(T) \neq \emptyset$ if and only if $S_0 \subseteq T$ (this can be shown in the very same way as in the proof of Proposition 4.24).

Taken an arbitrary $f \in F$, let us show that it is non-emptying. Fix a set $T \subseteq C$ such that $\alpha(T) \neq \emptyset$. This equivalently means that $S_0 \subseteq T$, so

$$\begin{aligned}
& \alpha(T) \supseteq \alpha(S_0) = S_0 \\
\implies & f(\alpha(T)) \supseteq f(S_0) \\
\implies & f^A(\alpha(T)) = \alpha(f(\alpha(T))) \supseteq \alpha(f(S_0))
\end{aligned}$$

But $f(S_0) \in B_F$, so $\alpha(f(S_0)) = f(S_0) \neq \emptyset$ by the hypothesis that f is total, and so $f^A(\alpha(T)) \neq \emptyset$, from which we conclude that f is non-emptying. □

Proof of Theorem 4.28. Fix an N such that all $f \in F_N$ are non-emptying in A_N.

Define

$$\begin{aligned} E &= \{\tilde{c} \in C_N \mid \tilde{c} \notin R_N(c_0), \exists f_{\tilde{c}} \in F_N \ . \ f_{\tilde{c}}(\tilde{c}) = c_0\} \\ &= P_{F_N}(c_0) \setminus R_N(c_0). \end{aligned}$$

Now fix a function $f \in F_N$, and let $J(f)$ be the set of $\tilde{c}$ for which f can play the role of $f_{\tilde{c}}$, namely

$$J(f) = \{\tilde{c} \in C_N \setminus R_N(c_0) \mid f(\tilde{c}) = c_0\}.$$

By hypothesis (2), $|J(f)| \leq k_2(N)$.

Now let G be the set of functions in F_N that can play the role of $f_{\tilde{c}}$ for some $\tilde{c} \in E$:

$$G = \{f \in F_N \mid \exists \tilde{c} \in E \ . \ f(\tilde{c}) = c_0\}.$$

Clearly

$$E = \bigcup_{f \in G} J(f).$$

But we know that $|J(f)| \leq k_2(N)$ for all f, so

$$|E| \leq \sum_{f \in G} |J(f)| \leq |G| \cdot k_2(N).$$

By Lemma 4.7, for all $\tilde{c} \in E$ we have $f_{\tilde{c}}(c_0)$ is representable. This can be equivalently restated saying that for all $f \in G$, $f(c_0)$ is representable. So consider the set I of all possible images of c_0 through functions in G:

$$I = \{f(c_0) \mid f \in G\}.$$

This set is a subset of R_N because all its elements are representable.

Clearly,

$$G = \bigcup_{d \in I} \{f \in G \mid f(c_0) = d\}.$$

Now observe that, for any $d \in C$, by hypothesis (1) we have

$$|\{f \in G \mid f(c_0) = d\}| \leq k_1(N)$$

so

$$|G| \leq \sum_{d \in I} |\{f \in G \mid f(c_0) = d\}| \leq |I| \cdot k_1(N)$$

that in turn implies

$$|E| \leq |G| \cdot k_2(N) \leq |I| \cdot k_1(N) \cdot k_2(N).$$

Since I is a subset of R_N, we get

$$|E| \leq |I| \cdot k_1(N) \cdot k_2(N) \leq |R_N| \cdot k_1(N) \cdot k_2(N).$$

Lastly, we recall that $E = P_{F_N}(c_0) \setminus R_N(c_0)$. Therefore, if all $f \in F_N$ are non-emptying in A_N, then

$$|P_{F_N}(c_0)| \leq |E| \leq |R_N| \cdot k_1(N) \cdot k_2(N).$$

The last hypothesis of the theorem states that $|P_{F_N}(c_0)| = \omega(\log(N) \cdot k_1(N) \cdot k_2(N))$. Lemma 4.12 states that $|R_N| = O(\log(N))$. Therefore, there exists an N_0 such that the inequality

$$\omega(\log(N) \cdot k_1(N) \cdot k_2(N)) = |P_{F_N}(c_0)| \leq |R_N| \cdot k_1(N) \cdot k_2(N) = O(\log(N) \cdot k_1(N) \cdot k_2(N))$$

does not hold for any $N > N_0$. This in turn implies that for all $N > N_0$ it is not possible that all the functions $f \in F_N$ are non-emptying in A_N. □

Appendix C

Logics comparison supplementary materials

This appendix contains technical details of proofs and examples for Chapter 5.

Proof of Lemma 5.1. In the proof, we assume Q to be any set of states, and $\sigma' \in Q$ to be any of its elements.
Case $\llbracket \overleftarrow{\mathsf{r}_1; \mathsf{r}_2} \rrbracket$
By (5.2), $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1; \mathsf{r}_2} \rrbracket \sigma'$ if and only if $\sigma' \in \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket \sigma$.

$$\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket \sigma = \llbracket \mathsf{r}_2 \rrbracket (\llbracket \mathsf{r}_1 \rrbracket \sigma) = \bigcup_{\sigma'' \in \llbracket \mathsf{r}_1 \rrbracket \sigma} \llbracket \mathsf{r}_2 \rrbracket \sigma''$$

so $\sigma' \in \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket \sigma$ if and only if there exists a $\sigma'' \in \llbracket \mathsf{r}_1 \rrbracket \sigma$ such that $\sigma' \in \llbracket \mathsf{r}_2 \rrbracket \sigma''$. Again by (5.2), these are equivalent to $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \sigma''$ and $\sigma'' \in \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma'$, respectively. Hence

$$\sigma' \in \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket \sigma \iff \exists \sigma'' \in \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma' \,.\, \sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \sigma''$$

Since $\llbracket \overleftarrow{\cdot} \rrbracket$ is defined on sets by union

$$\llbracket \overleftarrow{\mathsf{r}_1} \rrbracket (\llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma') = \bigcup_{\sigma'' \in \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma'} \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \sigma''$$

which means $\exists \sigma'' \in \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma' \,.\, \sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \sigma''$ if and only if $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket (\llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma')$. Putting everything together, we get $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1; \mathsf{r}_2} \rrbracket \sigma'$ if and only if $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket (\llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma')$, so the two are the same set. The thesis follows easily lifting the equality by union on $\sigma' \in Q$ and by the arbitrariness of Q.
Case $\llbracket \overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2} \rrbracket$
By (5.2), $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2} \rrbracket \sigma'$ if and only if $\sigma' \in \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket \sigma$.

$$\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket \sigma = \llbracket \mathsf{r}_1 \rrbracket \sigma \cup \llbracket \mathsf{r}_2 \rrbracket \sigma$$

so $\sigma' \in \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket \sigma$ if and only if $\exists i \in \{1, 2\}$ such that $\sigma' \in \llbracket \mathsf{r}_i \rrbracket \sigma$. This is again equivalent to $\sigma \in \llbracket \overleftarrow{\mathsf{r}_i} \rrbracket \sigma'$, and

$$\exists i \in \{1, 2\} \,.\, \sigma \in \llbracket \overleftarrow{\mathsf{r}_i} \rrbracket \sigma' \iff \sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \sigma' \cup \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma'$$

Putting everything together, we get $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2} \rrbracket \sigma'$ if and only if $\sigma \in \llbracket \overleftarrow{\mathsf{r}_1} \rrbracket \sigma' \cup \llbracket \overleftarrow{\mathsf{r}_2} \rrbracket \sigma'$, which implies the thesis as in point 1.

Case $\llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket$
To prove this last equality, we define r^n inductively as the sequential composition of r with itself n times: $\mathsf{r}^1 = \mathsf{r}$ and $\mathsf{r}^{n+1} = \mathsf{r}^n; \mathsf{r}$. Clearly $\llbracket \mathsf{r}^n \rrbracket = \llbracket \mathsf{r} \rrbracket^n$. For simplicity, we also define $\llbracket \mathsf{r}^0 \rrbracket = \llbracket \overleftarrow{\mathsf{r}^0} \rrbracket = \llbracket \mathsf{r} \rrbracket^0$. We prove by induction on n that $\llbracket \overleftarrow{\mathsf{r}^n} \rrbracket = \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n$. For $n = 1$ we have $\llbracket \overleftarrow{\mathsf{r}^1} \rrbracket = \llbracket \overleftarrow{\mathsf{r}} \rrbracket^1$. If we assume it holds for n we have

$$\begin{aligned}
\llbracket \overleftarrow{\mathsf{r}^{n+1}} \rrbracket &= \llbracket \overleftarrow{\mathsf{r}^n; \mathsf{r}} \rrbracket && [\text{def. of } \mathsf{r}^n] \\
&= \llbracket \overleftarrow{\mathsf{r}^n} \rrbracket \circ \llbracket \overleftarrow{\mathsf{r}} \rrbracket && [\text{pt. 1 of this lemma}] \\
&= \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n \circ \llbracket \overleftarrow{\mathsf{r}} \rrbracket && [\text{inductive hp}] \\
&= \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n+1}
\end{aligned}$$

We then observe that

$$\begin{aligned}
\llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket \sigma' &= \{\sigma \mid \sigma' \in \llbracket \mathsf{r}^\star \rrbracket \sigma\} && [\text{def. of } \llbracket \overleftarrow{\cdot} \rrbracket] \\
&= \{\sigma \mid \sigma' \in \bigcup_{n \geq 0} \llbracket \mathsf{r} \rrbracket^n \sigma\} && [\text{def. of } \llbracket \mathsf{r}^\star \rrbracket] \\
&= \bigcup_{n \geq 0} \{\sigma \mid \sigma' \in \llbracket \mathsf{r} \rrbracket^n \sigma\} \\
&= \bigcup_{n \geq 0} \{\sigma \mid \sigma' \in \llbracket \mathsf{r}^n \rrbracket \sigma\} && [\text{observed above}] \\
&= \bigcup_{n \geq 0} \{\sigma \mid \sigma \in \llbracket \overleftarrow{\mathsf{r}^n} \rrbracket \sigma'\} && [(5.2)] \\
&= \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}^n} \rrbracket \sigma' \\
&= \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n \sigma' && [\text{shown above}]
\end{aligned}$$

As in the cases above, the thesis follows. □

Proof of Proposition 5.3. By definition of $\llbracket \overleftarrow{\cdot} \rrbracket$ we have

$$\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q = \bigcup_{\sigma' \in Q} \{\sigma \mid \sigma \in \llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma'\} = \{\sigma \mid \exists \sigma' \in Q \,.\, \sigma' \in \llbracket \mathsf{r} \rrbracket \sigma\}$$

Using this,

$$P \subseteq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \iff \forall \sigma \in P \,.\, \sigma \in \{\sigma \mid \exists \sigma' \in Q \,.\, \sigma' \in \llbracket \mathsf{r} \rrbracket \sigma\} \iff \forall \sigma \in P \,.\, \exists \sigma' \in Q \,.\, \sigma' \in \llbracket \mathsf{r} \rrbracket \sigma$$

□

We split the proof between soundness and completeness.

Proposition C.1 (SIL is sound)**.** *Any provable SIL triple is valid.*

Proof. The proof is by structural induction on the derivation tree.
Case $\langle\!\langle \mathsf{atom} \rangle\!\rangle$
This case is trivial since $\llbracket \overleftarrow{\mathsf{c}} \rrbracket Q \subseteq \llbracket \overleftarrow{\mathsf{c}} \rrbracket Q$.

Case $\langle\!\langle\mathsf{cons}\rangle\!\rangle$
We have that

$$P \subseteq P' \subseteq \llbracket\overleftarrow{\mathsf{r}}\rrbracket Q' \subseteq \llbracket\overleftarrow{\mathsf{r}}\rrbracket Q$$

The inequalities above are justified, in order, by the hypothesis of the rule, by the inductive hypothesis on $\langle\!\langle P'\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q'\rangle\!\rangle$, by monotonicity of $\llbracket\overleftarrow{\mathsf{r}}\rrbracket$ and the hypothesis of the rule.
Case $\langle\!\langle\mathsf{seq}\rangle\!\rangle$
We have that

$$P \subseteq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket R \subseteq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q = \llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket Q$$

The inequalities above are justified, in order, by the inductive hypothesis on $\langle\!\langle P\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle R\rangle\!\rangle$, by the inductive hypothesis on $\langle\!\langle R\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q\rangle\!\rangle$, by Lemma 5.1.
Case $\langle\!\langle\mathsf{choice}\rangle\!\rangle$
We have that

$$P_1 \cup P_2 \subseteq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket Q \cup \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q = \llbracket\overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2}\rrbracket Q$$

The (in)equalities above are justified, in order, by the two inductive hypotheses, by Lemma 5.1.
Case $\langle\!\langle\mathsf{iter}\rangle\!\rangle$
We first prove by induction on n that $Q_n \subseteq \llbracket\overleftarrow{\mathsf{r}}\rrbracket^n Q_0$. The base case $n = 0$ is trivial because $Q_0 \subseteq \llbracket\overleftarrow{\mathsf{r}}\rrbracket^0 Q_0$. For the inductive case, we have

$$Q_{n+1} \subseteq \llbracket\overleftarrow{\mathsf{r}}\rrbracket Q_n \subseteq \llbracket\overleftarrow{\mathsf{r}}\rrbracket\llbracket\overleftarrow{\mathsf{r}}\rrbracket^n Q_0 = \llbracket\overleftarrow{\mathsf{r}}\rrbracket^{n+1} Q_0$$

The (in)equalities above are justified, in order, by inductive hypothesis on $\langle\!\langle Q_{n+1}\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_n\rangle\!\rangle$, the inductive hypothesis for n and monotonicity of $\llbracket\overleftarrow{\mathsf{r}}\rrbracket$, by definition of $\llbracket\overleftarrow{\mathsf{r}}\rrbracket^{n+1}$.

With this, we prove

$$\bigcup_{n\geq 0} Q_n \subseteq \bigcup_{n\geq 0}\llbracket\overleftarrow{\mathsf{r}}\rrbracket^n Q_0 = \llbracket\overleftarrow{\mathsf{r}}\rrbracket^\star Q_0$$

The (in)equalities above are justified, in order, by the proof above and by Lemma 5.1. □

Proposition C.2 (SIL is complete)**.** *Any valid SIL triple is provable.*

Proof. First we show that, for any Q, the triple $\langle\!\langle\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle$ is provable by induction on the structure of r.
Case $\mathsf{r} = \mathsf{c}$
We can prove $\langle\!\langle\llbracket\overleftarrow{\mathsf{c}}\rrbracket Q\rangle\!\rangle\ \mathsf{c}\ \langle\!\langle Q\rangle\!\rangle$ using $\langle\!\langle\mathsf{atom}\rangle\!\rangle$.
Case $\mathsf{r} = \mathsf{r}_1;\mathsf{r}_2$
We can prove $\langle\!\langle\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_1;\mathsf{r}_2\ \langle\!\langle Q\rangle\!\rangle$ with

$$\frac{\langle\!\langle\llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q\rangle\!\rangle \quad \langle\!\langle\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle Q\rangle\!\rangle}{\langle\!\langle\llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_1;\mathsf{r}_2\ \langle\!\langle Q\rangle\!\rangle}\ \langle\!\langle\mathsf{seq}\rangle\!\rangle$$

where the two premises can be proved by inductive hypothesis, and $\llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket Q = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q$ by Lemma 5.1.
Case $\mathsf{r} = \mathsf{r}_1 \oplus \mathsf{r}_2$
We can prove $\langle\!\langle\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle Q\rangle\!\rangle$ with

$$\frac{\forall i \in \{1,2\} \quad \langle\!\langle\llbracket\overleftarrow{\mathsf{r}_i}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_i\ \langle\!\langle Q\rangle\!\rangle}{\langle\!\langle\llbracket\overleftarrow{\mathsf{r}_1}\rrbracket Q \cup \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q\rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle Q\rangle\!\rangle}\ \langle\!\langle\mathsf{choice}\rangle\!\rangle$$

where the two premises can be proved by inductive hypothesis, and $\llbracket\overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2}\rrbracket Q = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket Q \cup \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q$ by Lemma 5.1.

Case $\mathsf{r} = \mathsf{r}^\star$
We can prove $\langle\!\langle \llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket Q \rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle Q \rangle\!\rangle$ with

$$\frac{\forall n \geq 0 \,.\, \langle\!\langle \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n+1} Q \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n} Q \rangle\!\rangle}{\langle\!\langle \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n} Q \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle}\ \langle\!\langle \mathsf{iter} \rangle\!\rangle$$

where the premises can be proved by inductive hypothesis since $\llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n+1} Q = \llbracket \overleftarrow{\mathsf{r}} \rrbracket \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n} Q$, and $\llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket Q = \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n} Q$ by Lemma 5.1.

To conclude the proof, take a triple $\langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$ such that $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \supseteq P$. Then we can first prove the triple $\langle\!\langle \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$, and then using rule $\langle\!\langle \mathsf{cons} \rangle\!\rangle$ we derive $\langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$. □

The proof of Theorem 5.4 is a corollary of Proposition C.1–C.2.

Proof of Proposition 5.5. The proof is by structural induction on the derivation tree and extends that of Proposition C.1 with inductive cases for the new rules.
Case $\langle\!\langle \mathsf{empty} \rangle\!\rangle$
Clearly $\emptyset \subseteq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$.
Case $\langle\!\langle \mathsf{disj} \rangle\!\rangle$
We have that

$$P_1 \cup P_2 \subseteq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q_1 \cup \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q_2 = \llbracket \overleftarrow{\mathsf{r}} \rrbracket (Q_1 \cup Q_2)$$

The (in)equalities above are justified, in order, by inductive hypotheses on $\langle\!\langle P_1 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_1 \rangle\!\rangle$ and $\langle\!\langle P_2 \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q_2 \rangle\!\rangle$, by additivity of $\llbracket \overleftarrow{\mathsf{r}} \rrbracket$.
Case $\langle\!\langle \mathsf{iter0} \rangle\!\rangle$
We have that

$$Q = \llbracket \overleftarrow{\mathsf{r}} \rrbracket^0 Q \subseteq \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n Q = \llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket Q$$

The last equality above is justified by Lemma 5.1.
Case $\langle\!\langle \mathsf{unroll} \rangle\!\rangle$
We have that

$$P \subseteq \llbracket \overleftarrow{\mathsf{r}^\star ; \mathsf{r}} \rrbracket Q = \llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q = \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n (\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q) = \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^{n+1} Q \subseteq \bigcup_{n \geq 0} \llbracket \overleftarrow{\mathsf{r}} \rrbracket^n Q = \llbracket \overleftarrow{\mathsf{r}^\star} \rrbracket Q$$

The first inequality is justified by the inductive hypothesis on $\langle\!\langle P \rangle\!\rangle\ \mathsf{r}^\star ; \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$, other equalities are justified by Lemma 5.1. □

Proof of Proposition 5.9. We prove the left-to-right implication, so assume $\llbracket \mathsf{r} \rrbracket P \subseteq Q$. Take a state $\sigma' \in \neg Q$. This means $\sigma' \notin Q$, that implies $\sigma' \notin \llbracket \mathsf{r} \rrbracket P$. So, for any state $\sigma \in P$, we have $\sigma' \notin \llbracket \mathsf{r} \rrbracket \sigma$, which is equivalent to $\sigma \notin \llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma'$ by (5.2). This being true for all $\sigma \in P$ means $P \cap \llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma' = \emptyset$, that is equivalent to $\llbracket \overleftarrow{\mathsf{r}} \rrbracket \sigma' \subseteq \neg P$. Since this holds for all states $\sigma' \in \neg Q$, we have $\llbracket \overleftarrow{\mathsf{r}} \rrbracket (\neg Q) \subseteq \neg P$.

The other implication is analogous. □

Proof of Proposition 5.13. To prove the first point, assume $\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q \supseteq P$ and take $\sigma' \in \llbracket \mathsf{r} \rrbracket P$. Then there exists $\sigma \in P$ such that $\sigma' \in \llbracket \mathsf{r} \rrbracket \sigma$. Since r is deterministic, $\llbracket \mathsf{r} \rrbracket \sigma$ can contain at most one element, hence $\llbracket \mathsf{r} \rrbracket \sigma = \{\sigma'\}$. Moreover, since $\sigma \in P \subseteq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$ there must exists a $\sigma'' \in Q$ such that $\sigma'' \in \llbracket \mathsf{r} \rrbracket \sigma = \{\sigma'\}$, which means $\sigma' \in Q$. Again, by arbitrariness of $\sigma' \in \llbracket \mathsf{r} \rrbracket P$, this implies $\llbracket \mathsf{r} \rrbracket P \subseteq Q$.

To prove the second point, assume $\llbracket \mathsf{r} \rrbracket P \subseteq Q$ and take a state $\sigma \in P$. Since r is terminating, $\llbracket \mathsf{r} \rrbracket \sigma$ is not empty, hence we can take $\sigma' \in \llbracket \mathsf{r} \rrbracket \sigma$. The hypothesis $\llbracket \mathsf{r} \rrbracket P \subseteq Q$

implies that $\sigma' \in Q$. Then, by (5.2), $\sigma \in [\![\overleftarrow{\mathsf{r}}]\!]\sigma' \subseteq [\![\overleftarrow{\mathsf{r}}]\!]Q$. By arbitrariness of $\sigma \in P$, this implies $P \subseteq [\![\overleftarrow{\mathsf{r}}]\!]Q$. □

Proof of Lemma 5.12. We first prove that $[\![\overleftarrow{\mathsf{r}}]\!][\![\mathsf{r}]\!]P \supseteq P \setminus D_{\mathsf{r}}$. Take a $\sigma \in P \setminus D_{\mathsf{r}}$. Because $\sigma \notin D_{\mathsf{r}}$, $[\![\mathsf{r}]\!]\sigma \neq \emptyset$, so take $\sigma' \in [\![\mathsf{r}]\!]\sigma$. Since $\sigma \in P$ we have $\sigma' \in [\![\mathsf{r}]\!]P$. Moreover, by (5.2), we get $\sigma \in [\![\overleftarrow{\mathsf{r}}]\!]\sigma' \subseteq [\![\mathsf{r}]\!]P$. By arbitrariness of $\sigma \in P \setminus D_{\mathsf{r}}$ we have the thesis.

The proof for $[\![\mathsf{r}]\!][\![\overleftarrow{\mathsf{r}}]\!]Q \supseteq Q \setminus U_{\mathsf{r}}$ is analogous. □

Proof of Proposition 5.15. By definition, $[\![\mathsf{r}]\!]$ is additive, that is $[\![\mathsf{r}]\!](P_1 \cup P_2) = [\![\mathsf{r}]\!]P_1 \cup [\![\mathsf{r}]\!]P_2$. Take all P such that $[\![\mathsf{r}]\!]P \subseteq Q$. By additivity of $[\![\mathsf{r}]\!]$, their union satisfies the same inequality, hence it is the weakest such P.

By definition, $[\![\overleftarrow{\mathsf{r}}]\!]$ is additive. Analogously, take all Q such that $[\![\overleftarrow{\mathsf{r}}]\!]Q \subseteq P$. By additivity of $[\![\overleftarrow{\mathsf{r}}]\!]$, their union is the weakest Q satisfying that inequality. □

Proof of Proposition 5.16. The proof is given by the counterexamples in Example 5.17. For IL, the example shows that for $x = 1$ there is no strongest P such that $[\![\mathsf{r}]\!]P \supseteq (x = 1)$: $x = 0$ and $x = 10$ are incomparable and are both minimal, as $\emptyset$ is not a valid precondition. The argument for SIL is analogous with precondition $x = 1$. □

C.1 Proofs about Separation SIL

Given two stores $s, s' \in$ Store and a heap command $\mathsf{r} \in \mathsf{HRCmd}$, we use the notation $s \dot{\sim}_{\mathsf{r}} s'$ to indicate that they coincide on all variables not modified by r: $\forall x \notin \mathrm{mod}(\mathsf{r}) \,.\, s(x) = s'(x)$. Please note that $\dot{\sim}_{\mathsf{r}}$ is an equivalence relation.

Lemma C.3. *Let $(s, h) \in \Sigma$, $\mathsf{r} \in \mathsf{HRCmd}$. If $(s', h') \in [\![\mathsf{r}]\!](s, h)$ then $s \dot{\sim}_{\mathsf{r}} s'$.*

Proof. The proof is by induction on the syntax of r. We prove here only some relevant cases.

Case `x := a`

$(s', h') \in [\![\texttt{x := a}]\!](s, h)$ means that $s' = s[x \mapsto (\!|\mathsf{a}|\!)s]$. Particularly, this means that for all variables $y \neq x$, $s'(y) = s(y)$, which is the thesis because $\mathrm{mod}(\texttt{x := a}) = \{x\}$.

Case `free(x)`

$(s', h') \in [\![\texttt{free(x)}]\!](s, h)$ means that $s' = s$, which is the thesis because $\mathrm{mod}(\texttt{free(x)}) = \emptyset$.

Case $\mathsf{r}_1; \mathsf{r}_2$

$(s', h') \in [\![\mathsf{r}_1; \mathsf{r}_2]\!](s, h)$ means that there exists $(s'', h'') \in [\![\mathsf{r}_1]\!](s, h)$ such that $(s', h') \in [\![\mathsf{r}_2]\!](s'', h'')$. By inductive hypothesis, since $\mathrm{mod}(\mathsf{r}_1) \subseteq \mathrm{mod}(\mathsf{r}_1; \mathsf{r}_2)$, we have $s'' \dot{\sim}_{\mathsf{r}_1;\mathsf{r}_2} s$. Analogously, $\mathrm{mod}(\mathsf{r}_2) \subseteq \mathrm{mod}(\mathsf{r}_1; \mathsf{r}_2)$ implies $s' \dot{\sim}_{\mathsf{r}_1;\mathsf{r}_2} s''$. From these, we get $s' \dot{\sim}_{\mathsf{r}_1;\mathsf{r}_2} s$. □

The next technical proposition states some semantic properties of the assertion language to be exploited in the proof of Lemma 5.18.

Proposition C.4. *Let $p \in Asl$, $s, s' \in Store$, $h \in Heap$ and $a \in AExp$.*

1. *If $\forall x \in fv(p) \,.\, s(x) = s'(x)$ and $(s, h) \in \{\!|p|\!\}$ then $(s', h) \in \{\!|p|\!\}$.*

2. *If $(s, h) \in \{\!|p[a/x]|\!\}$ then $(s[x \mapsto (\!|a|\!)s], h) \in \{\!|p|\!\}$.*

Proof. By structural induction on the syntax of assertions. □

Proof of Lemma 5.18. First, we observe that Proposition 5.3 does not depend on the specific definition of $[\![\cdot]\!]$, thus it holds for separation SIL as well. Thanks to this, we prove the thesis through the equivalent condition

$$\forall (s,h) \in \{\!|p * t|\!\} \,.\, \exists (s',h') \in \{\!|q * t|\!\} \,.\, (s',h') \in [\![\mathsf{r}]\!](s,h)$$

The proof is by induction on the derivation tree of the provable triple $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q\rangle\!\rangle$. We prove here only some relevant cases.

Case $\langle\!\langle\mathsf{assign}\rangle\!\rangle$

Take $(s,h) \in \{\!|q[a/x] * t|\!\}$. Then we can split $h = h_p \bullet h_t$ such that $(s,h_p) \in \{\!|q[a/x]|\!\}$ and $(s,h_t) \in \{\!|t|\!\}$. Let $s' = s[x \mapsto (\!|\mathsf{a}|\!)s]$, so that $(s',h) \in [\![\texttt{x := a}]\!]\{\!|q[a/x] * t|\!\}$. Since $\mathrm{fv}(t) \cap \mathrm{mod}(\mathsf{r}) = \emptyset$, $x \notin \mathrm{fv}(t)$. Thus, by Proposition C.4.1, $(s',h_t) \in \{\!|t|\!\}$. Moreover, $(s',h_p) \in \{\!|q|\!\}$ by Proposition C.4.2. Hence, $(s',h_p \bullet h_t) = (s',h) \in \{\!|q * t|\!\}$.

Case $\langle\!\langle\mathsf{alloc}\rangle\!\rangle$

Take $(s,h) \in \{\!|\mathbf{emp} * t|\!\} = \{\!|t|\!\}$. Take a location $l \notin \mathrm{dom}(h)$, and let $s' = s[x \mapsto l]$, $h' = h[l \mapsto s(v)]$, so that $(s',h') \in [\![\texttt{x := alloc()}]\!](s,h)$. We can split $h' = [l \mapsto s(v)] \bullet h$ because $l \notin \mathrm{dom}(h)$. Since $\mathrm{fv}(t) \cap \mathrm{mod}(\mathsf{r}) = \emptyset$, $x \notin \mathrm{fv}(t)$. Thus, by Proposition C.4.1, $(s',h) \in \{\!|t|\!\}$. Moreover, $(s',[l \mapsto s(v)]) = (s',[s'(x) \mapsto s'(v)])$, which satisfies $(s',[s'(x) \mapsto s'(v)]) \in \{\!|x \mapsto v|\!\}$. Hence $(s',h') \in \{\!|x \mapsto v * t|\!\}$.

Case $\langle\!\langle\mathsf{load}\rangle\!\rangle$

Take $(s,h) \in \{\!|y \mapsto a * q[a/x] * t|\!\}$. Then we know $x \notin \mathrm{fv}(t)$ and $h = [s(y) \mapsto (\!|a|\!)s] \bullet h_p \bullet h_t$, $(s,h_p) \in \{\!|q[a/x]|\!\}$, $(s,h_t) \in \{\!|t|\!\}$. Let $s' = s[x \mapsto h(s(y))] = s[x \mapsto (\!|a|\!)s]$. By Proposition C.4.1, $(s',h_t) \in \{\!|t|\!\}$. By Proposition C.4.2, $(s',h_p) \in \{\!|q|\!\}$. Lastly, since $x \notin \mathrm{fv}(a)$, $(\!|a|\!)s' = (\!|a|\!)s$ and $s'(y) = s(y)$, so we have $(s',[s'(y) \mapsto (\!|a|\!)s']) \in \{\!|y \mapsto a|\!\}$. Combining these, $(s',h) = (s',[s(y) \mapsto (\!|a|\!)s] \bullet h_p \bullet h_t) \in \{\!|y \mapsto a * q * t|\!\}$. The thesis follows observing that $(s',h) \in [\![\texttt{x := [y]}]\!](s,h)$.

Case $\langle\!\langle\mathsf{store}\rangle\!\rangle$

Take $(s,h) \in \{\!|x \mapsto - * t|\!\}$. Then $x \notin \mathrm{fv}(t)$ and exists $v \in \mathrm{Val}$ such that $h = [s(x) \mapsto v] \bullet h_t$, $(s,h_t) \in \{\!|t|\!\}$. Let $h' = h[s(x) \mapsto s(y)]$. Clearly $h' = [s(x) \mapsto s(y)] \bullet h_t$ and $(s,[s(x) \mapsto s(y)]) \in \{\!|x \mapsto y|\!\}$. Hence $(s,h') \in \{\!|x \mapsto y * t|\!\}$ and $(s,h') \in [\![[x] := y]\!](s,h)$, which is the thesis.

Case $\langle\!\langle\mathsf{exists}\rangle\!\rangle$

Take $(s,h) \in \{\!|(\exists x.p) * t|\!\}$. Then there exists a value $v \in \mathrm{Val}$ and decomposition $h = h_p \bullet h_t$ such that $(s[x \mapsto v],h_p) \in \{\!|p|\!\}$ and $(s,h_t) \in \{\!|t|\!\}$. Without loss of generality, we can assume $x \notin \mathrm{fv}(t)$; otherwise, we just rename it using a fresh name neither in t nor in r. Hence, by Proposition C.4.1, $(s[x \mapsto v],h_t) \in \{\!|t|\!\}$. So $(s[x \mapsto v],h) \in \{\!|p * t|\!\}$. By inductive hypothesis on the provable triple $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q\rangle\!\rangle$ and formula t, there is $(s',h') \in \{\!|q * t|\!\}$ such that $(s',h') \in [\![\mathsf{r}]\!](s[x \mapsto v],h)$. Because $x \notin \mathrm{fv}(\mathsf{r})$, we also have $(s',h') \in [\![\mathsf{r}]\!](s,h)$, and clearly $(s',h') \in \{\!|(\exists x.q) * t|\!\}$, that is the thesis.

Case $\langle\!\langle\mathsf{frame}\rangle\!\rangle$

By hypothesis, $(\mathrm{fv}(t' * t)) \cap \mathrm{mod}(\mathsf{r}) = (\mathrm{fv}(t') \cup \mathrm{fv}(t)) \cap \mathrm{mod}(\mathsf{r}) = \emptyset$. Then, applying the inductive hypothesis on the provable triple $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q\rangle\!\rangle$ and the formula $t' * t$ (which satisfies the hypothesis of the theorem) we get exactly the thesis.

Case $\langle\!\langle\mathsf{seq}\rangle\!\rangle$

Because of name clashes, here we assume the hypotheses of rule $\langle\!\langle\mathsf{seq}\rangle\!\rangle$ to be $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle p'\rangle\!\rangle$ and $\langle\!\langle p'\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$. Since $\mathrm{mod}(r_1) \cup \mathrm{mod}(r_2) = \mathrm{mod}(r_1; r_2)$, we know that $\mathrm{fv}(t) \cap \mathrm{mod}(r_1) = \mathrm{fv}(t) \cap \mathrm{mod}(r_2) = \emptyset$. Take $(s,h) \in \{\!|p * t|\!\}$. By inductive hypothesis on provable triple $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle p'\rangle\!\rangle$ and formula t we get that there exists $(s'',h'') \in \{\!|p' * t|\!\}$ such that $(s'',h'') \in [\![\mathsf{r}_1]\!](s,h)$. Then, by inductive hypothesis on the provable triple $\langle\!\langle p'\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$ and formula t again, we get $(s',h') \in \{\!|q * t|\!\}$ such that $(s',h') \in [\![\mathsf{r}_2]\!](s'',h'')$. The thesis follows since $(s',h') \in [\![\mathsf{r}_2]\!](s'',h'') \subseteq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!](s,h)) = [\![\mathsf{r}_1; \mathsf{r}_2]\!](s,h)$. □

Proof of Theorem 5.21. The proof follows the same line as the soundness proof of “plain” Separation SIL. For notation, we write (ϵ, s, h) for state $(\epsilon, (s, h))$ from domain $\{ok, er\} \times \Sigma$, where ϵ is the flag. Following the proof of Lemma 5.18, we prove by induction on the derivation tree of $\langle\!\langle \epsilon : p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle \epsilon : q \rangle\!\rangle$ the condition

$$\forall (\epsilon, s, h) \in \{\!|\epsilon : p * t|\!\} \,.\, \exists (\epsilon', s', h') \in \{\!|\epsilon' : q * t|\!\} \,.\, (\epsilon', s', h') \in [\![\mathsf{r}]\!](\epsilon, s, h)$$

We prove here only some relevant cases.

Case $\langle\!\langle$assign$\rangle\!\rangle$

Rule $\langle\!\langle$assign$\rangle\!\rangle$ requires both flags ϵ and ϵ' to be ok. Take $(ok, s, h) \in \{\!|ok : p * t|\!\}$. Since the semantics of assignments $(\!|\mathsf{x\ :=\ a}|\!)$ never fails, on ok states it behaves exactly as in the Separation SIL model without flags. Therefore, the proof concludes as in Lemma 5.18.

Case $\langle\!\langle$frame$\rangle\!\rangle$

By hypothesis, $(\mathrm{fv}(t' * t)) \cap \mathrm{mod}(\mathsf{r}) = (\mathrm{fv}(t') \cup \mathrm{fv}(t)) \cap \mathrm{mod}(\mathsf{r}) = \emptyset$. Then, applying the inductive hypothesis on the provable triple $\langle\!\langle \epsilon : p \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle \epsilon' : q \rangle\!\rangle$ and the formula $t' * t$ (which satisfies the hypothesis of the theorem) we get exactly the thesis.

Case $\langle\!\langle$seq$\rangle\!\rangle$

Because of name clashes, we let the hypotheses of rule $\langle\!\langle$seq$\rangle\!\rangle$ be $\langle\!\langle \epsilon : p \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle \epsilon' : p' \rangle\!\rangle$ and $\langle\!\langle \epsilon' : p' \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle \epsilon'' : q \rangle\!\rangle$. Since $\mathrm{mod}(r_1) \cup \mathrm{mod}(r_2) = \mathrm{mod}(r_1; r_2)$, we know that $\mathrm{fv}(t) \cap \mathrm{mod}(r_1) = \mathrm{fv}(t) \cap \mathrm{mod}(r_2) = \emptyset$. Take $(\epsilon, s, h) \in \{\!|\epsilon : p * t|\!\}$. By inductive hypothesis on provable triple $\langle\!\langle \epsilon : p \rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle \epsilon' : p' \rangle\!\rangle$ and formula t we get that there exists $(\epsilon', s', h') \in \{\!|\epsilon' : p' * t|\!\}$ such that $(\epsilon', s', h') \in [\![\mathsf{r}_1]\!](\epsilon, s, h)$. Then, by inductive hypothesis on the provable triple $\langle\!\langle \epsilon' : p' \rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle \epsilon'' : q \rangle\!\rangle$ and formula t again, we get $(\epsilon'', s'', h'') \in \{\!|\epsilon'' : q * t|\!\}$ such that $(\epsilon'', s'', h'') \in [\![\mathsf{r}_2]\!](\epsilon', s', h')$. The thesis follows since $(\epsilon'', s'', h'') \in [\![\mathsf{r}_2]\!](\epsilon', s', h') \subseteq [\![\mathsf{r}_2]\!]([\![\mathsf{r}_1]\!](\epsilon, s, h)) = [\![\mathsf{r}_1; \mathsf{r}_2]\!](\epsilon, s, h)$.

Case $\langle\!\langle$store-er$\rangle\!\rangle$

Take $(ok, s, h) \in \{\!|ok : x \not\mapsto\ * t|\!\}$. $h(s(x)) = \delta \notin \mathrm{Val}$, so that $(er, s, h) \in [\![[x] := y]\!](ok, s, h)$. Moreover, since $(s, h) \in \{\!|x \not\mapsto * t|\!\}$, we have $(er, s, h) \in \{\!|er : x \not\mapsto\ * t|\!\}$, which is the thesis.

Case $\langle\!\langle$er-id$\rangle\!\rangle$

Take $(er, s, h) \in \{\!|er : q|\!\}$. Since $[\![\cdot]\!]$ always acts as the identity on er states, we have $(er, s, h) \in [\![\mathsf{r}]\!](er, s, h)$ and $(er, s, h) \in \{\!|er : q|\!\}$. □

Some of the following equivalences are standard, but we collect them all here for convenience.

Lemma C.5. *For all assertions p_1, p_2, q, variables $x, x' \in Var$ and arithmetic expressions $a_1, a_2 \in \mathsf{AExp}$, the following equivalences hold:*

1. $(p_1 \vee p_2) \wedge q \equiv (p_1 \wedge q) \vee (p_2 \wedge q)$
2. $(p_1 \vee p_2) * q \equiv (p_1 * q) \vee (p_2 * q)$
3. $\exists x.(p \vee q) \equiv (\exists x.p) \vee (\exists x.q)$
4. $\exists x.(p \wedge q) \equiv (\exists x.p) \wedge q \quad$ *if $x \notin fv(q)$*
5. $\exists x.(p * q) \equiv (\exists x.p) * q \quad$ *if $x \notin fv(q)$*
6. $a_1 \asymp a_2 \wedge (p_1 * p_2) \equiv (a_1 \asymp a_2 \wedge p_1) * p_2$
7. $(p_1 * x \mapsto x') \wedge (p_2 * x \mapsto x') \equiv (p_1 \wedge p_2) * x \mapsto x'$
8. $(p_1 * x \not\mapsto\ ) \wedge (p_2 * x \not\mapsto\ ) \equiv (p_1 \wedge p_2) * x \not\mapsto$

Proof. Point (1), (3) and (4) are standard in first-order logic. Point (2), (5) and (6) are standard in separation logic [Rey02].

For point (7) we observe

$$
\begin{aligned}
&\{\!|(p_1 * x \mapsto x') \land (p_2 * x \mapsto x')|\!\}\\
&= \{(s,h) \mid (s,h) \in \{\!|p_1 * x \mapsto x'|\!\}, (s,h) \in \{\!|p_2 * x \mapsto x'|\!\}\}\\
&= \{(s,h) \mid h = h_1 \bullet [s(x) \mapsto s(x')], (s,h_1) \in \{\!|p_1|\!\} h = h_2 \bullet [s(x) \mapsto s(x')], (s,h_2) \in \{\!|p_2|\!\}\}\\
&= \{(s,h) \mid h = h' \bullet [s(x) \mapsto s(x')], (s,h') \in \{\!|p_1|\!\}, (s,h') \in \{\!|p_2|\!\}\}\\
&= \{\!|x \mapsto x' * (p_1 \land p_2)|\!\}
\end{aligned}
$$

Point (8) is analogous. □

Lemma C.6. *Let* $\mathsf{c} \in \mathsf{HACmd}$*,* $z \notin fv(\mathsf{c})$ *be a fresh variable, and* $(s,h), (s',h') \in \Sigma$ *be two states such that* $(s,h) \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket (s',h')$*. Then, for any value* $v \in Val$*,*

$$(s[z \mapsto v], h) \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket (s'[z \mapsto v], h')$$

Proof. By definition of $\llbracket \overleftarrow{\mathsf{c}} \rrbracket$, we know that $(s',h') \in (\!|\mathsf{c}|\!)(s,h)$ and that the thesis is equivalent to $(s'[z \mapsto v], h') \in (\!|\mathsf{c}|\!)(s[z \mapsto v], h)$. The proof is by cases on c.

Case `skip`

Since $(\!|\texttt{skip}|\!)$ is the identity, $(s,h) = (s',h')$. Therefore, $(s'[z \mapsto v], h') \in (\!|\texttt{skip}|\!)(s[z \mapsto v], h)$.

Case `x := a`

By definition of $(\!|\texttt{x := a}|\!)$, $s' = s[x \mapsto (\!|a|\!)s]$ and $h' = h$. Since $z \notin \text{fv}(\texttt{x := a})$, $z \neq x$ and $z \notin \text{fv}(a)$, therefore

$$s[z \mapsto v][x \mapsto (\!|a|\!)(s[z \mapsto v])] = s[z \mapsto v][x \mapsto (\!|a|\!)s] = s[x \mapsto (\!|a|\!)s][z \mapsto v] = s'[z \mapsto v]$$

With this, we have $(s'[z \mapsto v], h') \in (\!|\texttt{x := a}|\!)(s[z \mapsto v], h)$.

Case `free(x)`

By definition of $(\!|\texttt{free(x)}|\!)$, $s' = s$, $h' = h[s(x) \mapsto \delta]$. Since $z \notin \text{fv}(\texttt{free(x)})$ then $z \neq x$. With this, we have $(s'[z \mapsto v], h') \in (\!|\texttt{free(x)}|\!)(s[z \mapsto v], h)$. □

Proof of Lemma 5.24. The proof is by induction on the structure of q.

Case $q = \textbf{false}$

Take $p = \textbf{false}$.

Case $q = \textbf{true}$

Take $p = \textbf{true}$.

Case $q = q_1 \land q_2$

We consider point (1) first. By inductive hypothesis, there exists p_1 and p_2 such that

$$
\begin{aligned}
q_1 \land (x \mapsto x' * \textbf{true}) \equiv x \mapsto x' * p_1\\
q_2 \land (x \mapsto x' * \textbf{true}) \equiv x \mapsto x' * p_2
\end{aligned}
$$

so that

$$
\begin{aligned}
q \land (x \mapsto x' * \textbf{true}) &\equiv q_1 \land q_2 \land (x \mapsto x' * \textbf{true})\\
&\equiv q_1 \land (x \mapsto x' * \textbf{true}) \land q_2 \land (x \mapsto x' * \textbf{true})\\
&\equiv (x \mapsto x' * p_1) \land (x \mapsto x' * p_2)\\
&\equiv x \mapsto x' * (p_1 \land p_2)
\end{aligned}
$$

where we used Lemma C.5.7 for the last equivalence. The case for point (2) is analogous using Lemma C.5.8 instead.
Case $q = q_1 \vee q_2$
We consider point (1) first. By inductive hypothesis, there exists p_1 and p_2 such that

$$\begin{aligned} q_1 \wedge (x \mapsto x' * \mathbf{true}) &\equiv x \mapsto x' * p_1 \\ q_2 \wedge (x \mapsto x' * \mathbf{true}) &\equiv x \mapsto x' * p_2 \end{aligned}$$

so that

$$\begin{aligned} q \wedge (x \mapsto x' * \mathbf{true}) &\equiv (q_1 \vee q_2) \wedge (x \mapsto x' * \mathbf{true}) \\ &\equiv (q_1 \wedge (x \mapsto x' * \mathbf{true})) \vee (q_2 \wedge (x \mapsto x' * \mathbf{true})) \\ &\equiv (x \mapsto x' * p_1) \vee (x \mapsto x' * p_2) \\ &\equiv x \mapsto x' * (p_1 \vee p_2) \end{aligned}$$

where we used Lemma C.5.1 for the second equivalence and Lemma C.5.2 for the last one. The case for point (2) is analogous.
Case $q = a_1 \asymp a_2$
Both points follow from Lemma C.5.6 by taking $p_1 = \mathbf{true}$ and $p_2 = x \mapsto x'$ (resp. $p_2 = x \not\mapsto$).
Case $q = \mathbf{emp}$
Both formulae $\mathbf{emp} \wedge (x \mapsto x' * \mathbf{true})$ and $\mathbf{emp} \wedge (x \not\mapsto * \mathbf{true})$ are not satisfiable. Therefore we get the thesis with $p = \mathbf{false}$.
Case $q = z \mapsto z'$
For point (1):

$$\begin{aligned} &\{\!| z \mapsto z' \wedge (x \mapsto x' * \mathbf{true}) |\!\} \\ &= \{ (s, h) \mid (s, h) \in \{\!| z \mapsto z' |\!\}, (s, h) \in \{\!| x \mapsto x' * \mathbf{true} |\!\} \} \\ &= \{ (s, h) \mid h = [s(z) \mapsto s(z')], h = [s(x) \mapsto s(x')] \bullet h_t \} \\ &= \{ (s, h) \mid h = [s(x) \mapsto s(x')], s(z) = s(x), s(z') = s(x') \} \\ &= \{\!| z' = x' \wedge z = x \wedge x \mapsto x' |\!\} \end{aligned}$$

For point (2), we observe that $z \mapsto z' \wedge (x \not\mapsto * \mathbf{true})$ is not satisfiable, so we get the thesis with $p = \mathbf{false}$.
Case $q = z \not\mapsto$
For point (1), we observe that $z \not\mapsto \wedge (x \mapsto x' * \mathbf{true})$ is not satisfiable, so we get the thesis with $p = \mathbf{false}$.

For point (2):

$$\begin{aligned} &\{\!| z \not\mapsto \wedge (x \not\mapsto * \mathbf{true}) |\!\} \\ &= \{ (s, h) \mid (s, h) \in \{\!| z \not\mapsto |\!\}, (s, h) \in \{\!| x \not\mapsto * \mathbf{true} |\!\} \} \\ &= \{ (s, h) \mid h = [s(z) \mapsto \delta], h = [s(x) \mapsto \delta] \bullet h_t \} \\ &= \{ (s, h) \mid h = [s(x) \mapsto \delta], s(z) = s(x) \} \\ &= \{\!| z = x \wedge x \not\mapsto |\!\} \end{aligned}$$

Case $q = q_1 * q_2$
We consider point (1) first. By inductive hypothesis, there exists p_1 and p_2 such that

$$\begin{aligned} q_1 \wedge (x \mapsto x' * \mathbf{true}) &\equiv x \mapsto x' * p_1 \\ q_2 \wedge (x \mapsto x' * \mathbf{true}) &\equiv x \mapsto x' * p_2 \end{aligned}$$

Take $p = p_1 * q_2 \lor q_1 * p_2$. We have

$$\{\!|x \mapsto x' * p|\!\} = \{\!|x \mapsto x' * p_1 * q_2|\!\} \cup \{\!|x \mapsto x' * q_1 * p_2|\!\}$$

Now consider

$$\begin{aligned}
&\{\!|q \land (x \mapsto x' * \mathbf{true})|\!\} \\
&= \{\!|(q_1 * q_2) \land (x \mapsto x' * \mathbf{true})|\!\} \\
&= \{(s,h) \mid (s,h) \in \{\!|x \mapsto x' * \mathbf{true}|\!\}, h = h_1 \bullet h_2, (s,h_1) \in \{\!|q_1|\!\}, (s,h_2) \in \{\!|q_2|\!\}\} \\
&= \{(s,h) \mid h(s(x)) = s(x'), h = h_1 \bullet h_2, (s,h_1) \in \{\!|q_1|\!\}, (s,h_2) \in \{\!|q_2|\!\}\}
\end{aligned}$$

For every state (s,h) in this set, either $s(x) \in \mathrm{dom}(h_1)$ or $s(x) \in \mathrm{dom}(h_2)$: it can't be in neither because $h(s(x)) = s(x')$. Consider the former case: then $(s,h_1) \in \{\!|x \mapsto x' * \mathbf{true}|\!\}$, so that $(s,h_1) \in \{\!|q_1 \land (x \mapsto x' * \mathbf{true})|\!\} = \{\!|x \mapsto x' * p_1|\!\}$, so that $(s, h_1 \bullet h_2) \in \{\!|x \mapsto x' * p_1 * q_2|\!\}$. Analogously, in the latter case $(s, h_1 \bullet h_2) \in \{\!|x \mapsto x' * q_1 * p_2|\!\}$. Therefore, $\{\!|q \land (x \mapsto x' * \mathbf{true})|\!\} \subseteq \{\!|x \mapsto x' * p|\!\}$.

For the other inclusion, consider

$$\begin{aligned}
&\{\!|x \mapsto x' * p_1 * q_2|\!\} \\
&= \{(s,h) \mid h = h_1 \bullet h_2, (s,h_1) \in \{\!|x \mapsto x' * p_1|\!\}, (s,h_2) \in \{\!|q_2|\!\}\} \\
&= \{(s,h) \mid h = h_1 \bullet h_2, (s,h_1) \in \{\!|q_1 \land (x \mapsto x' * \mathbf{true})|\!\}, (s,h_2) \in \{\!|q_2|\!\}\} \\
&= \{(s,h) \mid h = h_1 \bullet h_2, h_1(s(x)) = s(x'), (s,h_1) \in \{\!|q_1|\!\}, (s,h_2) \in \{\!|q_2|\!\}\} \\
&\subseteq \{(s,h) \mid h = h_1 \bullet h_2, h(s(x)) = s(x'), (s,h_1) \in \{\!|q_1|\!\}, (s,h_2) \in \{\!|q_2|\!\}\} \\
&= \{\!|q \land (x \mapsto x' * \mathbf{true})|\!\}
\end{aligned}$$

and analogously for $\{\!|x \mapsto x' * q_1 * p_2|\!\} \subseteq \{\!|q \land (x \mapsto x' * \mathbf{true})|\!\}$.

The case for point (2) is analogous. □

Lemma C.7. *Let $q \in Asl$ be an assertion without $\lor$ and $\exists$, and let $\mathsf{c} \in \mathsf{HACmd}$. Then there exists $p \in Asl$ such that $\{\!|p|\!\} = \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\}$, and $\langle\!\langle p \rangle\!\rangle$ c $\langle\!\langle q \rangle\!\rangle$ is provable.*

Proof. We recall that

$$\llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\} = \{(s,h) \mid (\!|\mathsf{c}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset\}$$

and that $\mathbf{err} \notin \{\!|q|\!\}$ for any q. In the proof below, we will use the following equivalence: given a state (s,h) such that $s(h(x)) \in \mathrm{Val}$, $(s,h) \in \{\!|x \mapsto - * \mathbf{true}|\!\}$. Therefore, $(s,h) \in \{\!|q|\!\}$ if and only if $(s,h) \in \{\!|q \land (x \mapsto - * \mathbf{true})|\!\}$. Using Lemma 5.24.1, there exists a q' (which depends on q and x but not on (s,h)) such that this is true if and only if $(s,h) \in \{\!|\exists x'.(x \mapsto x' * q')|\!\}$. Analogously (using Lemma 5.24.2), if $s(h(x)) = \delta$, $(s,h) \in \{\!|q|\!\}$ if and only if $(s,h) \in \{\!|x \not\mapsto * q'|\!\}$ for some q'.

We now proceed by cases on the heap atomic command c.

Case `skip`

We have

$$\llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\} = \{(s,h) \mid (\!|\mathtt{skip}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset\} = \{(s,h) \mid (s,h) \in \{\!|q|\!\}\}$$

So we have the thesis taking $p = q$, and we prove it by using $\langle\!\langle\mathsf{skip}\rangle\!\rangle$ and $\langle\!\langle\mathsf{frame}\rangle\!\rangle$.

Case `x := a`

We have

$$\begin{aligned}
\llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\} &= \{(s,h) \mid (\!|\mathtt{x\ :=\ a}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset\} \\
&= \{(s,h) \mid (s[x \mapsto (\!|\mathsf{a}|\!)s], h) \in \{\!|q|\!\}\} \\
&= \{(s,h) \mid (s,h) \in \{\!|q[a/x]|\!\}\}
\end{aligned}$$

So we have the thesis taking $p = q[a/x]$, and we prove it by using $\langle\!\langle \mathsf{assign} \rangle\!\rangle$.

Case `b?`

We have

$$\begin{aligned}
[\![\overleftarrow{\mathsf{c}}]\!]\{\!|q|\!\} &= \{(s,h) \mid (\!|\mathtt{b?}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset\} \\
&= \{(s,h) \mid (\!|\mathtt{b}|\!)s = \mathtt{tt}, (s,h) \in \{\!|q|\!\}\} \\
&= \{\!|q \wedge b|\!\}
\end{aligned}$$

So we have the thesis taking $p = q \wedge b$, and we prove it by using $\langle\!\langle \mathsf{assume} \rangle\!\rangle$.

Case `x := alloc()`

We have

$$\begin{aligned}
[\![\overleftarrow{\mathsf{c}}]\!]\{\!|q|\!\} &= \{(s,h) \mid (\!|\mathtt{x\ :=\ alloc()}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset\} \\
&= \{(s,h) \mid \exists l, v. h(l) = \delta, (s[x \mapsto l], h[l \mapsto v]) \in \{\!|q|\!\}\} \\
&= \{(s,h) \mid \exists l, v. h(l) = \delta, (s[x \mapsto l], h[l \mapsto v]) \in \{\!|\exists x'. x \mapsto x' * q'|\!\}\} \\
&= \{(s,h) \mid \exists l, v. h(l) = \delta, \exists v'.(s[x \mapsto l][x' \mapsto v'], h[l \mapsto v]) \in \{\!|x \mapsto x' * q'|\!\}\} \\
&= \{(s,h) \mid \exists l, v. h = [l \mapsto \delta] \bullet h_q, \exists v'. \\
&\qquad\qquad (s[x \mapsto l][x' \mapsto v'], [l \mapsto v]) \in \{\!|x \mapsto x'|\!\}, \\
&\qquad\qquad (s[x \mapsto l][x' \mapsto v'], h_q) \in \{\!|q'|\!\}\} \\
&= \{(s,h) \mid \exists l, v. h = [l \mapsto \delta] \bullet h_q, (s[x \mapsto l][x' \mapsto v], h_q) \in \{\!|q'|\!\}\} \\
&= \{\!|\exists i. \exists x'. i \not\mapsto\ * q'[i/x]|\!\}
\end{aligned}$$

for fresh variables i. So we have the thesis taking $p = \exists i. \exists x'. i \not\mapsto\ * q'[i/x]$. To prove the triple $\langle\!\langle p \rangle\!\rangle$ `x := alloc()` $\langle\!\langle q \rangle\!\rangle$ we first observe the following chain of implications:

$$\begin{aligned}
q &\Longleftarrow \exists x'. x \mapsto x' * q' && \text{[Lemma 5.24.1]} \\
&\equiv \exists i. \exists x'. x \mapsto x' * q' && [i \text{ fresh}] \\
&\Longleftarrow \exists i. \exists x'. x = i \wedge (x \mapsto x' * q') && \\
&\Longleftarrow \exists i. \exists x'. x = i \wedge (x \mapsto x' * q'[i/x]) && \text{[replacing } i = x] \\
&\Longleftarrow \exists i. \exists x'. (x = i \wedge x \mapsto x') * q'[i/x] && \text{[Lemma C.5.6]}
\end{aligned}$$

Then we prove the triple with the following derivation tree:

$$\dfrac{\dfrac{\dfrac{x \notin \mathrm{fv}(q'[i/x]) \qquad \dfrac{}{\langle\!\langle i \not\mapsto\ \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle x = i \wedge x \mapsto x' \rangle\!\rangle}\ \langle\!\langle \mathsf{alloc} \rangle\!\rangle}{\langle\!\langle i \not\mapsto\ * q'[i/x] \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle (x = i \wedge x \mapsto x') * q'[i/x] \rangle\!\rangle}\ \langle\!\langle \mathsf{frame} \rangle\!\rangle}{\langle\!\langle p \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle \exists i. \exists x'. (x = i \wedge x \mapsto x') * q'[i/x] \rangle\!\rangle}\ \langle\!\langle \mathsf{exists} \rangle\!\rangle \text{ x2}}{\langle\!\langle p \rangle\!\rangle\ \mathsf{c}\ \langle\!\langle q \rangle\!\rangle}\ \langle\!\langle \mathsf{cons} \rangle\!\rangle$$

Case `free(x)`

We have

$$\begin{aligned}
[\![\overleftarrow{\mathsf{c}}]\!]\{\!|q|\!\} &= \{(s,h) \mid (\!|\mathtt{free(x)}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset\} \\
&= \{(s,h) \mid h(s(x)) \in \mathrm{Val}, (s, h[s(x) \mapsto \delta]) \in \{\!|q|\!\}\} \\
&= \{(s,h) \mid h(s(x)) \in \mathrm{Val}, (s, h[s(x) \mapsto \delta]) \in \{\!|x \not\mapsto\ * q'|\!\}\} \\
&= \{(s,h) \mid h(s(x)) \in \mathrm{Val}, h = [s(x) \mapsto h(s(x))] \bullet h_q, (s, h_q) \in \{\!|q'|\!\}\} \\
&= \{\!|x \mapsto - * q'|\!\}
\end{aligned}$$

So we have the thesis taking $p = x \mapsto - * q'$, and we prove it by using $\langle\!\langle \mathsf{free} \rangle\!\rangle$ and $\langle\!\langle \mathsf{frame} \rangle\!\rangle$ with frame q' (this is always possible because $\mathrm{mod}(\texttt{free(x)}) = \emptyset$).

Case `x := [y]`

We have

$$\begin{aligned}
\llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\} &= \{ (s,h) \mid (\!|\texttt{x := [y]}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset \} \\
&= \{ (s,h) \mid h(s(y)) \in \mathrm{Val}, (s[x \mapsto h(s(y))], h) \in \{\!|q|\!\} \} \\
&= \{ (s,h) \mid h(s(y)) \in \mathrm{Val}, (s[x \mapsto h(s(y))], h) \in \{\!|\exists y'. y \mapsto y' * q'|\!\} \} \\
&= \{ (s,h) \mid h(s(y)) \in \mathrm{Val}, h = [s(y) \mapsto h(s(y))] \bullet h_q, (s[x \mapsto h(s(y))], h_q) \in \{\!|q'|\!\} \} \\
&= \{\!|\exists y'.(y \mapsto y' * q'[y'/x])|\!\}
\end{aligned}$$

So we have the thesis taking $p = \exists y'.(y \mapsto y' * q'[y'/x])$, and we prove it by using $\langle\!\langle \mathsf{load} \rangle\!\rangle$ with $a = y'$ and $\langle\!\langle \mathsf{exists} \rangle\!\rangle$ because y' is fresh.

Case `[x] := y`

We have

$$\begin{aligned}
\llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\} &= \{ (s,h) \mid (\!|\texttt{[x] := y}|\!)(s,h) \cap \{\!|q|\!\} \neq \emptyset \} \\
&= \{ (s,h) \mid h(s(x)) \in \mathrm{Val}, (s, h[s(x) \mapsto s(y)]) \in \{\!|q|\!\} \} \\
&= \{ (s,h) \mid h(s(x)) \in \mathrm{Val}, (s, h[s(x) \mapsto s(y)]) \in \{\!|\exists x'. x \mapsto x' * q'|\!\} \} \\
&= \{ (s,h) \mid h(s(x)) \in \mathrm{Val}, h = [s(x) \mapsto h(s(x))] \bullet h_q, (s, h_q) \in \{\!|q'|\!\} \} \\
&= \{\!|x \mapsto - * q'|\!\}
\end{aligned}$$

So we have the thesis taking $p = x \mapsto - * q'$. To prove the triple $\langle\!\langle p \rangle\!\rangle$ `[x] := y` $\langle\!\langle q \rangle\!\rangle$, we first prove $\langle\!\langle p \rangle\!\rangle$ `[x] := y` $\langle\!\langle x \mapsto y * q' \rangle\!\rangle$ by using $\langle\!\langle \mathsf{store} \rangle\!\rangle$ and $\langle\!\langle \mathsf{frame} \rangle\!\rangle$ with frame q' (this is always possible because $\mathrm{mod}(\texttt{[x] := y}) = \emptyset$). Then we observe that $x \mapsto y \implies \exists x'. x \mapsto x'$, so that we get $\langle\!\langle p \rangle\!\rangle$ `[x] := y` $\langle\!\langle q \rangle\!\rangle$ by using $\langle\!\langle \mathsf{cons} \rangle\!\rangle$. □

Lemma C.8. *Let $p, q \in Asl$ be two assertions and* $\mathsf{c} \in \mathsf{HACmd}$ *such that* $\{\!|p|\!\} = \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\}$. *Then, for $z \in Var$ fresh,* $\{\!|\exists z.p|\!\} = \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|\exists z.q|\!\}$

Proof. In this proof, we use the following notation: given a state $(s,h) = \sigma \in \Sigma$ and a value $v \in \mathrm{Val}$, we denote with σ_v the state $(s[z \mapsto v], h)$, where we performed in the store s the substitution of value v for the variable z of the statement of the lemma. We prove the two inclusions separately.

To show that $\{\!|\exists z.p|\!\} \supseteq \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|\exists z.q|\!\}$, take $\sigma \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|\exists z.q|\!\}$. Then, by definition of $\llbracket \overleftarrow{\mathsf{c}} \rrbracket$, there exist $\sigma' \in \{\!|\exists z.q|\!\}$ such that $\sigma \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \sigma'$. By definition of $\{\!|\exists z.q|\!\}$, there exists a value $v \in \mathrm{Val}$ such that $\sigma'_v \in \{\!|q|\!\}$. Then, since z is fresh, by Lemma C.6 $\sigma_v \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \sigma'_v \subseteq \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\}$. Since by hypothesis $\{\!|p|\!\} = \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\}$ we have $\sigma_v \in \{\!|p|\!\}$, and thus $\sigma \in \{\!|\exists z.p|\!\}$.

To show that $\{\!|\exists z.p|\!\} \subseteq \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|\exists z.q|\!\}$, take $(s,h) = \sigma \in \{\!|\exists z.p|\!\}$. By definition of $\{\!|\exists z.p|\!\}$, there exists a value v such that $\sigma_v \in \{\!|p|\!\}$, and $\{\!|p|\!\} = \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|q|\!\}$ by hypothesis. By definition of $\llbracket \overleftarrow{\mathsf{c}} \rrbracket$, there exist $\sigma' \in \{\!|q|\!\}$ such that $\sigma_v \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \sigma'$. Let $w = s(z)$: clearly $\sigma = (\sigma_v)_w$. Moreover, $\sigma_v \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \sigma'$ and z is fresh, so by Lemma C.6 we have $\sigma = (\sigma_v)_w \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \sigma'_w$. Lastly, since $\sigma' \in \{\!|q|\!\}$ we have $\sigma'_w \in \{\!|\exists z.q|\!\}$, thus $\sigma \in \llbracket \overleftarrow{\mathsf{c}} \rrbracket \{\!|\exists z.q|\!\}$. □

Proof of Theorem 5.23. First we fix q and prove, by induction on the structure of r, that there exists $p \in \mathrm{Asl}$ such that $\{\!|p|\!\} = \llbracket \overleftarrow{\mathsf{r}} \rrbracket \{\!|q|\!\}$, and $\langle\!\langle p \rangle\!\rangle$ r $\langle\!\langle q \rangle\!\rangle$ is provable.

Case $\mathsf{r} = \mathsf{c}$

First, we transform q in a normal form: we rename all quantified variables to fresh names, and use Lemma C.5 (points 1-5) to lift disjunctions to the top, then existential quantifiers. Thus, without loss of generality, we can assume that q is a disjunction of existentially quantified formulae that don't contain $\vee$ and $\exists$. Moreover, if we have a proof for each one of these formulae without $\vee$ and $\exists$, we can combine them using rules $\langle\!\langle\mathsf{disj}\rangle\!\rangle$ and $\langle\!\langle\mathsf{exists}\rangle\!\rangle$ to get a proof for the original q, and by Lemma C.8 and additivity of $\llbracket\overleftarrow{\mathsf{c}}\rrbracket$ that is the weakest precondition for q. Therefore, again without loss of generality, we can consider only the case in which q does not contain $\vee$ and $\exists$. This case is exactly Lemma C.7, so we conclude the inductive step.

Case $\mathsf{r} = \mathsf{r}_1; \mathsf{r}_2$
By inductive hypothesis on r_2, we know that there exists an assertion $t \in \mathrm{Asl}$ such that $\{\!|t|\!\} = \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\{\!|q|\!\}$ and $\langle\!\langle t\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$ is provable. By inductive hypothesis on r_1, we know that there exists an assertion $p \in \mathrm{Asl}$ such that $\{\!|p|\!\} = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\{\!|t|\!\}$ and $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle t\rangle\!\rangle$ is provable. Now $\{\!|p|\!\} = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\{\!|t|\!\} = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\{\!|q|\!\} = \llbracket\overleftarrow{\mathsf{r}_1; \mathsf{r}_2}\rrbracket\{\!|q|\!\}$ and we can prove $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1; \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$ using $\langle\!\langle\mathsf{seq}\rangle\!\rangle$ and the two proofs given by the inductive hypothesis.

Case $\mathsf{r} = \mathsf{r}_1 \oplus \mathsf{r}_2$
For $i = 1, 2$, by inductive hypothesis on r_i we know that there exists an assertion $p_i \in \mathrm{Asl}$ such that $\{\!|p_i|\!\} = \llbracket\overleftarrow{\mathsf{r}_i}\rrbracket\{\!|q|\!\}$ and $\langle\!\langle p_i\rangle\!\rangle\ \mathsf{r}_i\ \langle\!\langle q\rangle\!\rangle$ is provable. Therefore $\{\!|p_1 \vee p_2|\!\} = \{\!|p_1|\!\} \cup \{\!|p_2|\!\} = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\{\!|q|\!\} \cup \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\{\!|q|\!\} = \llbracket\overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2}\rrbracket\{\!|q|\!\}$ and we can prove $\langle\!\langle p_1 \vee p_2\rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$ using $\langle\!\langle\mathsf{choice}\rangle\!\rangle$ and the two proofs given by the inductive hypothesis.

Now take any $p, q \in \mathrm{Asl}$ such that $\llbracket\overleftarrow{\mathsf{r}}\rrbracket\{\!|q|\!\} \supseteq \{\!|p|\!\}$. By the proof above we know that there exists p' such that $\llbracket\overleftarrow{\mathsf{r}}\rrbracket\{\!|q|\!\} = \{\!|p'|\!\}$ and $\langle\!\langle p'\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q\rangle\!\rangle$ is provable. Since $\{\!|p|\!\} \subseteq \{\!|p'|\!\}$, the implication $p \implies p'$ holds. Using the oracle for this implication we can prove the triple $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q\rangle\!\rangle$ using $\langle\!\langle\mathsf{cons}\rangle\!\rangle$ and the proof of $\langle\!\langle p'\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q\rangle\!\rangle$. □

Proof of Theorem 5.25. The proof is by induction on the structure of r.

Case $\mathsf{r} = \mathsf{c}$
This is a special case of completeness for loop-free programs (Theorem 5.23).

Case $\mathsf{r} = \mathsf{r}_1; \mathsf{r}_2$
By inductive hypothesis, given any σ'' such that $\sigma'' \in \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\sigma'$ there exists an assertion t such that $\langle\!\langle t\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$ is provable and $\sigma'' \in \{\!|t|\!\}$. Particularly, we can take a σ'' such that $\sigma \in \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\sigma''$: this exists because $\sigma \in \llbracket\overleftarrow{\mathsf{r}_1; \mathsf{r}_2}\rrbracket\sigma' = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\sigma'$ (by Lemma 5.1). Then, again by inductive hypothesis, we get the assertion p such that $\sigma \in \{\!|p|\!\}$ and $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle t\rangle\!\rangle$ is provable. We conclude the inductive case by proving $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1; \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle$ via rule $\langle\!\langle\mathsf{seq}\rangle\!\rangle$.

Case $\mathsf{r} = \mathsf{r}_1 \oplus \mathsf{r}_2$
Since $\sigma \in \llbracket\overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2}\rrbracket\sigma' = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\sigma' \cup \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\sigma'$ (by Lemma 5.1), there must exist an $i \in \{1, 2\}$ such that $\sigma \in \llbracket\overleftarrow{\mathsf{r}_i}\rrbracket\sigma'$. Without loss of generality, we assume $i = 1$. By inductive hypothesis, we get an assertion p such that $\sigma \in \{\!|p|\!\}$ and $\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle q\rangle\!\rangle$ is provable. We conclude the inductive case with the proof

$$\dfrac{\dfrac{\text{(induction)}}{\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1\ \langle\!\langle q\rangle\!\rangle} \qquad \dfrac{}{\langle\!\langle \mathbf{false}\rangle\!\rangle\ \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle}\,\langle\!\langle\mathsf{empty}\rangle\!\rangle}{\langle\!\langle p\rangle\!\rangle\ \mathsf{r}_1 \oplus \mathsf{r}_2\ \langle\!\langle q\rangle\!\rangle}\,\langle\!\langle\mathsf{choice}\rangle\!\rangle$$

Case $\mathsf{r} = \mathsf{r}^\star$
Since $\sigma \in \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket\sigma' = \bigcup_{n \geq 0} \llbracket\overleftarrow{\mathsf{r}}\rrbracket^n\sigma'$ (by Lemma 5.1), there must exist an $m \geq 0$ such that $\sigma \in \llbracket\overleftarrow{\mathsf{r}}\rrbracket^m\sigma'$. Therefore, there must exist a sequence of states $\{\sigma_i\}_{0 \leq i \leq m}$ such that $\sigma_0 = \sigma'$, $\sigma_m = \sigma$ and $\sigma_{i+1} \in \llbracket\overleftarrow{\mathsf{r}}\rrbracket\sigma_i$ for all $0 \leq i < m$. By inductive hypothesis, fixed $q_0 = q$, there exists a corresponding sequence of assertions $\{q_i\}_{0 \leq i \leq m}$ such that $\sigma_i \in \{\!|q_i|\!\}$ and $\langle\!\langle q_{i+1}\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q_i\rangle\!\rangle$ is provable for all $0 \leq i < m$. We take $p = q_m$ and conclude the inductive case with the proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle\!\langle q_m\rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q_m\rangle\!\rangle}\ \langle\!\langle\mathsf{iter0}\rangle\!\rangle}{\vdots}}{\langle\!\langle q_m\rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q_2\rangle\!\rangle}\ \langle\!\langle\mathsf{unroll}\rangle\!\rangle \qquad \dfrac{\text{(induction)}}{\langle\!\langle q_2\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q_1\rangle\!\rangle}}{\dfrac{\langle\!\langle q_m\rangle\!\rangle\ \mathsf{r}^\star;\mathsf{r}\ \langle\!\langle q_1\rangle\!\rangle}{\langle\!\langle q_m\rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q_1\rangle\!\rangle}\ \langle\!\langle\mathsf{unroll}\rangle\!\rangle}\ \langle\!\langle\mathsf{seq}\rangle\!\rangle \qquad \dfrac{\text{(induction)}}{\langle\!\langle q_1\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle q_0\rangle\!\rangle}}{\dfrac{\langle\!\langle q_m\rangle\!\rangle\ \mathsf{r}^\star;\mathsf{r}\ \langle\!\langle q_0\rangle\!\rangle}{\langle\!\langle q_m\rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle q_0\rangle\!\rangle}\ \langle\!\langle\mathsf{unroll}\rangle\!\rangle}\ \langle\!\langle\mathsf{seq}\rangle\!\rangle$$

□

Appendix D

LCL$_A$ supplementary materials

This appendix contains technical details of proofs and examples for Chapter 6.

Proof of Theorem 6.2, extensional soundness of LCL$_A$. First we remark that points (1) and (3) implies point (2):

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && \text{[(1) and monotonocity of } \alpha\text{]}\\
&\leq \llbracket \mathsf{r} \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A\text{]}\\
&= \alpha(Q) && \text{[(3)]}
\end{aligned}$$

So all the lines are equal, in particular $\alpha(Q) = \alpha(\llbracket \mathsf{r} \rrbracket P)$. The proof is then by induction on the derivation tree of $\vdash_A [P]\ \mathsf{r}\ [Q]$, but we only have to prove (1) and (3) because of the observation above.

Case (transfer)

(1) It follows since $\llbracket \mathsf{e} \rrbracket P \leq \llbracket \mathsf{e} \rrbracket P$.

(3) It follows from the local completeness hypothesis $\mathbb{C}^A_P(\llbracket \mathsf{e} \rrbracket)$: $\llbracket \mathsf{e} \rrbracket^A \alpha(P) = \alpha(\llbracket \mathsf{e} \rrbracket \gamma\alpha(P)) = \alpha(\llbracket \mathsf{e} \rrbracket P)$.

Case (relax)

(1) By inductive hypothesis we know $Q' \leq \llbracket \mathsf{r} \rrbracket P'$, and together with the other two hypotheses of the rule we have $Q \leq Q' \leq \llbracket \mathsf{r} \rrbracket P' \leq \llbracket \mathsf{r} \rrbracket P$.

(3) We remark that $\alpha(P) = \alpha(P')$ if and only if $A(P) = A(P')$ by injectivity of γ in a GI, and that $P' \leq P \leq A(P')$ implies $A(P) = A(P')$. Thus the hypotheses of the rule gives $\alpha(P) = \alpha(P')$ and $\alpha(Q) = \alpha(Q')$. Point (3) then follow by the inductive hypothesis $\llbracket \mathsf{r} \rrbracket^A \alpha(P') = \alpha(Q')$:

$$\llbracket \mathsf{r} \rrbracket^A \alpha(P) = \llbracket \mathsf{r} \rrbracket^A \alpha(P') = \alpha(Q') = \alpha(Q)$$

Case (seq)

(1) $Q \leq \llbracket \mathsf{r}_2 \rrbracket R \leq \llbracket \mathsf{r}_2 \rrbracket (\llbracket \mathsf{r}_1 \rrbracket P) = \llbracket \mathsf{r1}; \mathsf{r}_2 \rrbracket P$, where the inequalities follow from inductive hypotheses and monotonicity of $\llbracket \mathsf{r}_2 \rrbracket$.

(3) We recall that $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A \leq \llbracket \mathsf{r}_2 \rrbracket^A \llbracket \mathsf{r}_1 \rrbracket^A$.

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket P) && \text{[(1) and monotonicity of } \alpha\text{]}\\
&\leq \llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^A\text{]}\\
&\leq \llbracket \mathsf{r}_2 \rrbracket^A \llbracket \mathsf{r}_1 \rrbracket^A \alpha(P) && \text{[recalled above]}\\
&= \llbracket \mathsf{r}_2 \rrbracket^A \alpha(R) && \text{[inductive hp]}\\
&= \alpha(Q) && \text{[inductive hp]}
\end{aligned}$$

So all the lines are equal, in particular $\llbracket \mathsf{r}_1; \mathsf{r}_2 \rrbracket^A \alpha(P) = \alpha(Q)$.

Case (join)
(1) By inductive hypotheses, $Q_1 \leq \llbracket \mathsf{r}_1 \rrbracket P$ and $Q_2 \leq \llbracket \mathsf{r}_2 \rrbracket P$. Hence $Q_1 \vee Q_2 \leq \llbracket \mathsf{r}_1 \rrbracket P \vee \llbracket \mathsf{r}_2 \rrbracket P = \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket P$.
(3) We observe that

$$\begin{aligned}
\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^A \alpha(P) &= \alpha \llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket \gamma\alpha(P) \\
&= \alpha(\llbracket \mathsf{r}_1 \rrbracket \gamma\alpha(P) \sqcup \llbracket \mathsf{r}_2 \rrbracket \gamma\alpha(P)) \\
&= \alpha\llbracket \mathsf{r}_1 \rrbracket \gamma\alpha(P) \sqcup \alpha\llbracket \mathsf{r}_2 \rrbracket \gamma\alpha(P) \\
&= \llbracket \mathsf{r}_1 \rrbracket^A \alpha(P) \sqcup \llbracket \mathsf{r}_2 \rrbracket^A \alpha(P)
\end{aligned}$$

where we used additivity of α. Recalling that by inductive hypotheses, $\alpha(Q_1) = \llbracket \mathsf{r}_1 \rrbracket^A \alpha(P)$ and $\alpha(Q_2) = \llbracket \mathsf{r}_2 \rrbracket^A \alpha(P)$, we get

$$\begin{aligned}
\alpha(Q_1 \vee Q_2) &= \alpha(Q_1) \sqcup \alpha(Q_2) && [\alpha \text{ is additive}] \\
&= \llbracket \mathsf{r}_1 \rrbracket^A \alpha(P) \sqcup \llbracket \mathsf{r}_2 \rrbracket^A \alpha(P) && [\text{inductive hypotheses}] \\
&= (\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2 \rrbracket^A)\alpha(P) && [\text{observation above}]
\end{aligned}$$

Case (rec)
(1) First, we show that $\llbracket \mathsf{r}^\star \rrbracket R \leq \llbracket \mathsf{r}^\star \rrbracket P$ using the inductive hypothesis $R \leq \llbracket \mathsf{r} \rrbracket P$:

$$\begin{aligned}
\llbracket \mathsf{r}^\star \rrbracket R &= \bigsqcup_{n \geq 0} \llbracket \mathsf{r} \rrbracket^n R && [\text{definition}] \\
&\leq \bigsqcup_{n \geq 0} \llbracket \mathsf{r} \rrbracket^n \llbracket \mathsf{r} \rrbracket P && [\text{inductive hp on all operands}] \\
&= \bigsqcup_{n \geq 1} \llbracket \mathsf{r} \rrbracket^n P && [\text{renaming the index variable } n] \\
&\leq \bigsqcup_{n \geq 0} \llbracket \mathsf{r} \rrbracket^n P && [\text{adding an element to the lub}] \\
&= \llbracket \mathsf{r}^\star \rrbracket P && [\text{definition}]
\end{aligned}$$

Now we show (1):

$$\begin{aligned}
Q &\leq \llbracket \mathsf{r}^\star \rrbracket (P \vee R) && [\text{inductive hp}] \\
&= \llbracket \mathsf{r}^\star \rrbracket P \vee \llbracket \mathsf{r}^\star \rrbracket R && [\text{additivity of } \llbracket \mathsf{r}^\star \rrbracket] \\
&\leq \llbracket \mathsf{r}^\star \rrbracket P \vee \llbracket \mathsf{r}^\star \rrbracket P && [\text{observation above}] \\
&= \llbracket \mathsf{r}^\star \rrbracket P
\end{aligned}$$

(3)

$$\begin{aligned}
\llbracket \mathsf{r}^\star \rrbracket^A \alpha(P) &\leq \llbracket \mathsf{r}^\star \rrbracket^A \alpha(P \vee R) && [\text{monotonicity of } \llbracket \mathsf{r}^\star \rrbracket^A \alpha] \\
&= \alpha(Q) && [\text{inductive hp}] \\
&\leq \alpha(\llbracket \mathsf{r}^\star \rrbracket P) && [(1) \text{ and monotonicity of } \alpha] \\
&\leq \llbracket \mathsf{r}^\star \rrbracket^A \alpha(P) && [\text{soundness of } \llbracket \mathsf{r}^\star \rrbracket^A]
\end{aligned}$$

Hence all the lines are equal, and in particular $\llbracket \mathsf{r}^\star \rrbracket^A \alpha(P) = \alpha(Q)$.
Case $(\mathsf{iterate})$
(1) We observe that $P = \llbracket \mathsf{r} \rrbracket^0 P \leq \llbracket \mathsf{r}^\star \rrbracket P$. Moreover, by inductive hypothesis $Q \leq \llbracket \mathsf{r} \rrbracket P$ so also $Q \leq \llbracket \mathsf{r}^\star \rrbracket P$. So by definition of lub $P \vee Q \leq \llbracket \mathsf{r}^\star \rrbracket P$.

(3) First we show by induction on $n \geq 1$ that $\alpha[\![\mathsf{r}]\!]^n\gamma\alpha(P) \leq [\![\mathsf{r}]\!]^A\alpha(P)$. The base case $n = 1$ is trivial recalling that $[\![\mathsf{r}]\!]^A = \alpha[\![\mathsf{r}]\!]\gamma$. So suppose it is true for n and let us show it for $n + 1$:

$$\begin{aligned}
\alpha[\![\mathsf{r}]\!]^{n+1}\gamma\alpha(P) &\leq \alpha[\![\mathsf{r}]\!]\gamma\alpha[\![\mathsf{r}]\!]^n\gamma\alpha(P) && [\gamma\alpha \leq \text{id}] \\
&\leq [\![\mathsf{r}]\!]^A[\![\mathsf{r}]\!]^A\alpha(P) && [\text{inductive hp } \alpha[\![\mathsf{r}]\!]^n\gamma\alpha(P) \leq [\![\mathsf{r}]\!]^A\alpha(P)] \\
&= [\![\mathsf{r}]\!]^A\alpha(Q) && [\text{inductive hp } \vdash_A [P]\ \mathsf{r}\ [Q]] \\
&\leq [\![\mathsf{r}]\!]^A\alpha(P) && [\text{hypothesis of the rule } Q \leq A(P)]
\end{aligned}$$

For the last inequality, we used that in any GC $Q \leq A(P) = \gamma\alpha(P)$ if and only if $\alpha(Q) \leq \alpha(P)$. Now we can prove the following chain of inequalities:

$$\begin{aligned}
[\![\mathsf{r}^\star]\!]^A\alpha(P) &= \alpha \bigsqcup_{n\geq 0} [\![\mathsf{r}]\!]^n\gamma\alpha(P) && [\text{definition}] \\
&= \bigsqcup_{n\geq 0} \alpha[\![\mathsf{r}]\!]^n\gamma\alpha(P) && [\alpha \text{ is additive}] \\
&= \alpha[\![\mathsf{r}]\!]^0\gamma\alpha(P) \sqcup \left(\bigsqcup_{n\geq 1} \alpha[\![\mathsf{r}]\!]^n\gamma\alpha(P) \right) && [\text{splitting the lub}] \\
&\leq \alpha[\![\mathsf{r}]\!]^0\gamma\alpha(P) \sqcup \left(\bigsqcup_{n\geq 1} [\![\mathsf{r}]\!]^A\alpha(P) \right) && [\text{shown above by induction}] \\
&= \alpha[\![\mathsf{r}]\!]^0\gamma\alpha(P) \sqcup [\![\mathsf{r}]\!]^A\alpha(P) && [\text{lub of constant terms}] \\
&= \alpha(P) \sqcup [\![\mathsf{r}]\!]^A\alpha(P) && [[\![\mathsf{r}]\!]^0 = \text{id and } \alpha\gamma\alpha = \alpha] \\
&= \alpha(P) \sqcup \alpha(Q) && [\text{inductive hp}] \\
&= \alpha(P \vee Q) && [\alpha \text{ is additive}] \\
&\leq \alpha([\![\mathsf{r}^\star]\!]P) && [(1) \text{ and monotonicity of } \alpha] \\
&\leq [\![\mathsf{r}^\star]\!]^A\alpha(P) && [\text{soundness of } [\![\mathsf{r}^\star]\!]^A]
\end{aligned}$$

Thus all the lines above are equal, and in particular $[\![\mathsf{r}^\star]\!]^A\alpha(P) = \alpha(P \vee Q)$. □

This technical lemma is used in the following proofs.

Lemma D.1. *If $A' \preceq A$ then $A = AA' = A'A$*

Proof. Fix a concrete element $c \in C$. Since $A' \preceq A$ we have $c \leq A'(c) \leq A(c)$. Applying A, by monotonicity we get $A(c) \leq AA'(c) \leq AA(c) = A(c)$, where the last equality is idempotency of A. This means $A = AA'$. Now consider $A'A(c)$. Since A is a closure operator $A'A(c) \leq A(A'A(c))$. But we just showed $AA'(A(c)) = A(A(c)) = A(c)$. Lastly, since A' is a closure operator too, $A(c) \leq A'A(c)$. Hence $A(c) \leq A'A(c) \leq A(c)$, so $A(c) = A'A(c)$. □

We point out that, by injectivity of γ, this also means $\alpha\gamma'\alpha' = \alpha$.

Proof of Theorem 6.3, extensional soundness of rule (refine-ext). We recall that the intuitive premise $A[\![\mathsf{r}]\!]^{A'}A(P) = A(Q)$ of the rule formally is $\alpha\gamma'[\![\mathsf{r}]\!]^{A'}\alpha'A(P) = \alpha(Q)$. Since the proof of Theorem 6.2 is by induction, we extend it by just proving the new inductive case. We also remark that we only need to prove points (1) and (3), since they imply point (2).

Case (refine-ext)
(1) It's the same as point (1) of extensional soundness applied to $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$, since this conclusion does not depend on the abstract domain.
(3)

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && [(1) \text{ and monotonicity of } \alpha]\\
&\leq \llbracket \mathsf{r} \rrbracket^{A} \alpha(P) && [\text{soundness of } \llbracket \mathsf{r} \rrbracket^{A}]\\
&= \alpha \llbracket \mathsf{r} \rrbracket \gamma\alpha(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha' \llbracket \mathsf{r} \rrbracket \gamma'\alpha'\gamma\alpha(P) && [\text{Lemma D.1}]\\
&= \alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) && [\text{definition}]\\
&= \alpha(Q) && [\text{hypothesis of the rule}]
\end{aligned}$$

Hence all the lines are equal; in particular $\llbracket r \rrbracket^{A}\alpha(P) = \alpha(Q)$. □

Proof of Proposition 6.7, extensional soundness of derived refinement rules. We show that the hypotheses of both rules implies those of (refine-ext). This means than whenever we can apply the former we could also apply the latter, so that Theorem 6.3 ensures extensional soundness.

The first two hypotheses $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ and $A' \preceq A$ are shared among all three rules, so we only have to show $\alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) = \alpha(Q)$. We recall that, by extensional soundness, $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$ implies (1) $Q \leq \llbracket \mathsf{r} \rrbracket P$ and (3) $\llbracket \mathsf{r} \rrbracket^{A'} \alpha'(P) = \alpha'(Q)$.

For (refine-int), we observe

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && [Q \leq \llbracket \mathsf{r} \rrbracket P \text{ and monotonicity of } \alpha]\\
&\leq \llbracket \mathsf{r} \rrbracket^{A} \alpha(P) && [\text{soundness of } \llbracket \mathsf{r} \rrbracket^{A}]\\
&= \alpha \llbracket \mathsf{r} \rrbracket A(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha' \llbracket \mathsf{r} \rrbracket A' A(P) && [\text{Lemma D.1}]\\
&= \alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) && [\text{definition}]\\
&\leq \alpha\gamma' \llbracket \mathsf{r} \rrbracket^{\sharp}_{A'} \alpha' A(P) && [\llbracket \mathsf{r} \rrbracket^{A'} \leq \llbracket \mathsf{r} \rrbracket^{\sharp}_{A'}]\\
&= \alpha(Q) && [\text{Last hypothesis of the rule}]
\end{aligned}$$

Hence all the lines are equal, and in particular $\alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) = \alpha(Q)$.

For (refine-pre), we observe

$$\begin{aligned}
\alpha(Q) &\leq \alpha(\llbracket \mathsf{r} \rrbracket P) && [Q \leq \llbracket \mathsf{r} \rrbracket P \text{ and monotonicity of } \alpha]\\
&\leq \llbracket \mathsf{r} \rrbracket^{A} \alpha(P) && [\text{soundness of } \llbracket \mathsf{r} \rrbracket^{A}]\\
&= \alpha \llbracket \mathsf{r} \rrbracket A(P) && [\text{definition}]\\
&= \alpha \llbracket \mathsf{r} \rrbracket A'(P) && [\text{hp of the rule}]\\
&= \alpha\gamma'\alpha' \llbracket \mathsf{r} \rrbracket A'(P) && [\text{Lemma D.1}]\\
&= \alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha'(P) && [\text{definition}]\\
&= \alpha\gamma'\alpha'(Q) && [\text{extensional soundness (3)}]\\
&= \alpha(Q) && [\text{Lemma D.1}]
\end{aligned}$$

Hence all the lines are equal, and in particular $\alpha\gamma' \llbracket \mathsf{r} \rrbracket^{A'} \alpha' A(P) = \alpha(Q)$. □

$$\dfrac{\dfrac{\mathbb{C}^{\mathrm{Oct}}_{\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}}(\llbracket\texttt{y := y - 1}\rrbracket)}{\vdash_{\mathrm{Oct}} [\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}]\ \texttt{y := y - 1}\ [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}-1=\mathtt{y}]}\ (\mathsf{transfer}) \qquad \dfrac{\mathbb{C}^{\mathrm{Oct}}_{\mathtt{y}\in\{1;99\}\wedge\mathtt{x}-1=\mathtt{y}}(\llbracket\texttt{x := x - 1}\rrbracket)}{\vdash_{\mathrm{Oct}} [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}-1=\mathtt{y}]\ \texttt{x := x - 1}\ [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y}]}\ (\mathsf{transfer})}{(**)}\ (\mathsf{seq})$$

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathrm{Oct}}_{R_2}(\llbracket\texttt{(x > 1)?}\rrbracket)}{\vdash_{\mathrm{Oct}} [R_2]\ \texttt{(x > 1)?}\ [\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}]}\ (\mathsf{transfer}) \qquad \dfrac{(**)}{\vdash_{\mathrm{Oct}} [\mathtt{y}\in\{2;100\}\wedge\mathtt{x}=\mathtt{y}]\ \texttt{y := y - 1; x := x - 1}\ [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y}]}\ (\mathsf{seq})}{\vdash_{\mathrm{Oct}} [R_2]\ \mathsf{r}_i\ [\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y}] \qquad (\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y})\leq A(R_2)}\ (\mathsf{seq})}{(*)}\ (\mathsf{iterate})$$

$$\dfrac{\dfrac{\mathbb{C}^{\mathrm{Oct}}_{R}(\llbracket\texttt{x := y}\rrbracket)}{\vdash_{\mathrm{Oct}} [R]\ \texttt{x := y}\ [R_2]}\ (\mathsf{transfer}) \qquad \dfrac{\dfrac{(*)}{\vdash_{\mathrm{Oct}} [R_2]\ \mathsf{r}_i^\star\ [R_2\vee(\mathtt{y}\in\{1;99\}\wedge\mathtt{x}=\mathtt{y})]}\ (\mathsf{iterate}) \qquad \dfrac{\mathbb{C}^{\mathrm{Oct}}_{R_2}(\llbracket\texttt{(x <= 1)?}\rrbracket)}{\vdash_{\mathrm{Oct}} [R_2]\ \texttt{(x <= 1)?}\ [Q]}\ (\mathsf{transfer})}{\vdash_{\mathrm{Oct}} [R_2]\ \mathsf{r}_i^\star\texttt{; (x <= 1)?}\ [Q]}\ (\mathsf{seq})}{\vdash_{\mathrm{Oct}} [R]\ \mathsf{r}_2\ [Q]}\ (\mathsf{seq})$$

Figure D.1: Derivation of $\vdash_{\mathrm{Oct}} [R]\ \mathsf{r}_2\ [Q]$ for Example 6.9.

Example D.2 (Details of Example 6.9.). The full derivation of the triple $\vdash_{\mathrm{Oct}} [R]\ \mathsf{r}_2\ [Q]$ is shown in Figure D.1, rotated and split to fit the page. We call the command iterated with the Kleene star $\mathsf{r}_i \triangleq$ `(x > 1)?; y := y - 1; x := x - 1`, and we let $R_2 \triangleq (\mathtt{y} \in \{1; 2; 100\} \wedge \mathtt{x} = \mathtt{y})$. We also used the logical implication $R_2 \implies (\mathtt{y} \in \{1; 99\} \wedge \mathtt{x} = \mathtt{y})$, both explicitly and implicitly in the equivalence $R_2 \vee (\mathtt{y} \in \{1; 99\} \wedge \mathtt{x} = \mathtt{y}) \iff R_2$. ■

Example D.3. Similarly to Example 6.10, we present another triple which is sound but cannot be proved in LCL_A extended with (refine-pre). The key difference is that this triple does not rely on divergence, but only on actual (im)precisions in the abstract domain.

Consider the concrete domain $C = \mathcal{P}(\mathbb{Z})$ of integers, the abstract domain Int of intervals, the concrete initial states $P = \{-1, 1\}$ and commands

$$\mathsf{r}_1 \triangleq \texttt{x != 0?}$$
$$\mathsf{r}_2 \triangleq \texttt{x >= 0?}$$

Then, the triple $\vdash_{\mathrm{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [\{1\}]$ is sound but cannot be proved in LCL_A extended with (refine-pre).

Following the same line of reasoning in Example 6.10, we observe that all strict subset $P' \subset P$ are such that $\mathrm{Int}(P') \subset P$, and the same property holds for all refinements $A' \preceq \mathrm{Int}$. Again, this means that we cannot apply (relax) to change P: to do it, we would need a $P' \subset P$ such that $P \subseteq A'(P')$.

Let $f_1 = [\![\mathsf{r}_1]\!]$ and $f_2 = [\![\mathsf{r}_2]\!]$. Observe that in the concrete semantics $f_1(P) = P$ and $f_2(P) = \{1\}$. Inspecting the logic, to prove the triple $\vdash_{\mathrm{Int}} [P]\ \mathsf{r}_1; \mathsf{r}_2\ [\{1\}]$ we can only apply three rules: (relax), (refine-pre) or (seq). To apply rule (relax) we would need either an under-approximation P' of P with the same abstraction, that does not exist by the observation above, or an over-approximation of Q, that would be unsound since $Q = f(P)$. Hence we cannot apply (relax). Suppose to apply (refine-pre): any A' used in the rule should satisfy $A' \preceq \mathrm{Int}$ and $A'(P) = \mathrm{Int}(P)$. As we pointed out above, we cannot apply (relax) even after the domain refinement. Therefore, the only rule that can be applied is (seq). To do that, we must prove two triples $\vdash_{A'} [P]\ \mathsf{r}_1\ [R]$ and $\vdash_{A'} [R]\ \mathsf{r}_2\ [Q]$ for some R.

Irrespective of how we prove the first triple, by soundness (Theorem 6.2) we have $R \subseteq f_1(P) = P$ and $A'(R) = A'(f_1(P)) = A'(P)$. If $R \subset P$ then $A'(R) \subset P \subseteq A'(P)$, which is a contradiction. So $R = P$. Now we should prove a triple $\vdash_{A'} [P]\ \mathsf{r}_2\ [Q]$, but this is impossible since, by soundness, this would imply local completeness of $[\![\mathsf{r}_2]\!] = f_2$ on P in A', that does not hold:

$$\begin{aligned} A' f_2(P) &= A'(\{1\}) \\ &\subseteq \mathrm{Int}(\{1\}) = \{1\} \\ &\subset \{0, 1\} = f_2(\mathrm{Int}(P)) = f_2 A'(P) \\ &\subseteq A' f_2 A'(P) \end{aligned}$$

Observe that, if we add (refine-int) to the proof system, we can use it to change the domain to one where we can express P (for instance, the concrete domain $\mathcal{P}(\mathbb{Z})$ or the refinement $\mathrm{Int}_P \triangleq \mathrm{Int} \cup \{P\}$) to prove the triple by applying (seq) and then (transfer) on both subtrees, as shown in Figure D.2. The two local completeness proof obligations are

$$\dfrac{\dfrac{\dfrac{\mathbb{C}^{\mathrm{Int}_P}_P(\llbracket \texttt{x != 0?}\rrbracket)}{\vdash_{\mathrm{Int}_P} [P]\ \texttt{x != 0?}\ [P]}\,(\mathsf{transfer}) \qquad \dfrac{\mathbb{C}^{\mathrm{Int}_P}_P(\llbracket \texttt{x >= 0?}\rrbracket)}{\vdash_{\mathrm{Int}_P} [P]\ \texttt{x >= 0?}\ [Q]}\,(\mathsf{transfer})}{\vdash_{\mathrm{Int}_P} [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q] \quad \mathrm{Int}_P \preceq \mathrm{Int} \quad \mathrm{Int}(\llbracket \mathsf{r}\rrbracket^{\sharp}_{\mathrm{Int}_P}(\mathrm{Int}(P))) = \mathrm{Int}(Q)}\,(\mathsf{seq})}{\vdash_{\mathrm{Int}}\ [P]\ \mathsf{r}_1;\mathsf{r}_2\ [Q]}\,(\mathsf{refine\text{-}int})$$

Figure D.2: Derivation of $\vdash_{\mathrm{Int}} [P]\ \mathsf{r}\ [Q]$ for Example D.3.

trivially satisfied because $P \in \mathrm{Int}_P$, and the hypothesis of ($\mathsf{refine\text{-}int}$) is verified:

$$\begin{aligned}
\mathrm{Int}(\llbracket \mathsf{r}\rrbracket^{\sharp}_{\mathrm{Int}_P}\mathrm{Int}(P)) &= \mathrm{Int}(\llbracket \mathsf{r}_2\rrbracket^{\mathrm{Int}_P}\llbracket \mathsf{r}_1\rrbracket^{\mathrm{Int}_P}\mathrm{Int}(P)) \\
&= \mathrm{Int}(\llbracket \mathsf{r}_2\rrbracket^{\mathrm{Int}_P}\llbracket \texttt{x != 0?}^{\mathrm{Int}_P}\rrbracket P) \\
&= \mathrm{Int}(\llbracket \texttt{x >= 0?}\rrbracket P) \\
&= \mathrm{Int}([1;1]) = \mathrm{Int}(Q)
\end{aligned}$$

■

The following lemma is used in the subsequent proof.

Lemma D.4. *Let* $\mathsf{r} \in \mathsf{Reg}$ *be a regular command. If* $A \preceq A'$ *then* $\gamma\llbracket \mathsf{r}\rrbracket^{\sharp}_{A}\alpha \le \gamma'\llbracket \mathsf{r}\rrbracket^{\sharp}_{A'}\alpha'$.

Proof. The proof is by structural induction on r.
Case ($\mathsf{r} = \mathsf{e}$)
$\gamma\llbracket \mathsf{r}\rrbracket^{\sharp}_{A}\alpha = \gamma\alpha\llparenthesis \mathsf{e}\rrparenthesis\gamma\alpha = A\llparenthesis \mathsf{e}\rrparenthesis A \le A'\llparenthesis \mathsf{e}\rrparenthesis A' = \gamma'\llbracket \mathsf{r}\rrbracket^{\sharp}_{A'}\alpha'$.
Case ($\mathsf{r}_1;\mathsf{r}_2$)

$$\begin{aligned}
\gamma\llbracket \mathsf{r}_1;\mathsf{r}_2\rrbracket^{\sharp}_{A}\alpha &= \gamma\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A}\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A}\alpha && \text{[definition]} \\
&= \gamma\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A}\alpha\gamma\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A}\alpha && [\alpha\gamma = \mathrm{id}_A] \\
&\le \gamma'\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha'\gamma'\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A'}\alpha' && \text{[inductive hp]} \\
&= \gamma'\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A'}\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A'}\alpha' && [\alpha'\gamma' = \mathrm{id}_{A'}] \\
&= \gamma'\llbracket \mathsf{r}_1;\mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha' && \text{[definition]}
\end{aligned}$$

Case ($\mathsf{r}_1 \oplus \mathsf{r}_2$)

$$\begin{aligned}
\gamma\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2\rrbracket^{\sharp}_{A}\alpha &= \gamma(\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A}\alpha \sqcup \llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A}\alpha) && \text{[definition]} \\
&= \gamma(\alpha\gamma\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A}\alpha \sqcup \alpha\gamma\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A}\alpha) && [\alpha\gamma = \mathrm{id}_A] \\
&= \gamma\alpha(\gamma\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A}\alpha \sqcup \gamma\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A}\alpha) && [\alpha \text{ is additive}] \\
&\le A(\gamma'\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha' \sqcup \gamma'\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A'}\alpha') && \text{[inductive hp]} \\
&\le A'(\gamma'\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha' \sqcup \gamma'\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A'}\alpha') && [A \le A'] \\
&= \gamma'(\alpha'\gamma'\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha' \sqcup \alpha'\gamma'\llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A'}\alpha') && [\alpha' \text{ is additive}] \\
&= \gamma'(\llbracket \mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha' \sqcup \llbracket \mathsf{r}_1\rrbracket^{\sharp}_{A'}\alpha') && [\alpha'\gamma' = \mathrm{id}_{A'}] \\
&= \gamma'\llbracket \mathsf{r}_1 \oplus \mathsf{r}_2\rrbracket^{\sharp}_{A'}\alpha' && \text{[definition]}
\end{aligned}$$

Case ($\mathsf{r}^{\star}$)
First we prove by induction on $n \ge 0$ that $\gamma(\llbracket \mathsf{r}\rrbracket^{\sharp}_{A})^{n}\alpha \le \gamma'(\llbracket \mathsf{r}\rrbracket^{\sharp}_{A'})^{n}\alpha'$ using the inductive

hypothesis that $\gamma[\![\mathsf{r}]\!]^\sharp_A\alpha \leq \gamma'[\![\mathsf{r}]\!]^\sharp_{A'}\alpha'$. For $n = 0$ the inequality to prove reduces to $\gamma\mathrm{id}_A\alpha = A \leq A' = \gamma'\mathrm{id}_{A'}\alpha'$. So suppose it is true for n:

$$\begin{aligned}
\gamma([\![\mathsf{r}]\!]^\sharp_A)^{n+1}\alpha &= \gamma([\![\mathsf{r}]\!]^\sharp_A)^n[\![\mathsf{r}]\!]^\sharp_A\alpha && [\text{definition}]\\
&= \gamma([\![\mathsf{r}]\!]^\sharp_A)^n\alpha\gamma[\![\mathsf{r}]\!]^\sharp_A\alpha && [\gamma\alpha = \mathrm{id}_A]\\
&\leq \gamma'([\![\mathsf{r}]\!]^\sharp_{A'})^n\alpha'\gamma'[\![\mathsf{r}]\!]^\sharp_{A'}\alpha' && [\text{inductive hps}]\\
&= \gamma'([\![\mathsf{r}]\!]^\sharp_{A'})^{n+1}\alpha' && [\gamma'\alpha' = \mathrm{id}_{A'}]
\end{aligned}$$

Now we proceed with the proof of the structural inductive statement:

$$\begin{aligned}
\gamma[\![\mathsf{r}^\star]\!]^\sharp_A\alpha &= \gamma\left(\bigsqcup_{n\geq 0}([\![\mathsf{r}]\!]^\sharp_A)^n\alpha\right) && [\text{definition}]\\
&= \gamma\left(\bigsqcup_{n\geq 0}\alpha\gamma([\![\mathsf{r}]\!]^\sharp_A)^n\alpha\right) && [\alpha\gamma = \mathrm{id}_A]\\
&= \gamma\alpha\left(\bigsqcup_{n\geq 0}\gamma([\![\mathsf{r}]\!]^\sharp_A)^n\alpha\right) && [\alpha \text{ is additive}]\\
&\leq A\left(\bigsqcup_{n\geq 0}\gamma'([\![\mathsf{r}]\!]^\sharp_{A'})^n\alpha'\right) && [\text{statement above}]\\
&\leq A'\left(\bigsqcup_{n\geq 0}\gamma'([\![\mathsf{r}]\!]^\sharp_{A'})^n\alpha'\right) && [A \leq A']\\
&= \gamma'\left(\bigsqcup_{n\geq 0}\alpha'\gamma'([\![\mathsf{r}]\!]^\sharp_{A'})^n\alpha'\right) && [\alpha' \text{ is additive}]\\
&= \gamma'\left(\bigsqcup_{n\geq 0}([\![\mathsf{r}]\!]^\sharp_{A'})^n\alpha'\right) && [\alpha'\gamma' = \mathrm{id}_{A'}]\\
&= \gamma'[\![\mathsf{r}^\star]\!]^\sharp_{A'}\alpha' && [\text{definition}]
\end{aligned}$$

□

Proof of Theorem 6.12, intensional soundness of rule (simplify)*.* Since the proof of Theorem 3.7 in [BGGR21] is by induction, we extend it by just proving the new inductive case.

Case (simplify)

(1) It's the same as point (1) of intensional soundness applied to $\vdash_{A'} [P]\ \mathsf{r}\ [Q]$, since this conclusion doesn't depend on the abstract domain.

(2-3)

$$\begin{aligned}
A'(Q) &= A(Q) && \text{[hypothesis of the rule]} \\
&\leq A(\llbracket \mathsf{r} \rrbracket P) && \text{[(1) and monotonicity of } A\text{]} \\
&= \gamma\alpha(\llbracket \mathsf{r} \rrbracket P) && \text{[definition]} \\
&\leq \gamma \llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(P) && \text{[soundness of } \llbracket \mathsf{r} \rrbracket^{\sharp}_{A}\text{]} \\
&\leq \gamma' \llbracket \mathsf{r} \rrbracket^{\sharp}_{A'} \alpha'(P) && \text{[Lemma D.4]} \\
&= \gamma'\alpha'(Q) && \text{[Soundness of } \vdash_{A'} [P]\ \mathsf{r}\ [Q]\text{]} \\
&= A'(Q) && \text{[definition]}
\end{aligned}$$

Hence all the lines are equal; in particular $\gamma\alpha(Q) = \gamma\alpha(\llbracket r \rrbracket P)$ and $\gamma\llbracket r \rrbracket^{\sharp}_{A}\alpha(P) = \gamma\alpha(Q)$. Since γ is injective, we get points (2-3) of intensional soundness. □

Proof of Theorem 6.14, intrinsic incompleteness of LCL$_A$ with rule (simplify). The proof follows closely that of intrinsic incompleteness of LCL$_A$ [BGGR21, Theorem 5.12]. The Turing completeness hypothesis allows to define, for any store $c \in \Sigma$, three regular commands $\mathsf{r}_{c?}$, $\mathsf{r}_{\neg c?}$ and r_c such that, given any input $S \in C = \mathcal{P}(\Sigma)$

$$\begin{aligned}
\llbracket \mathsf{r}_{c?} \rrbracket S &= S \cap \{c\} \\
\llbracket \mathsf{r}_{\neg c?} \rrbracket S &= S \setminus \{c\} \\
\llbracket \mathsf{r}_{c} \rrbracket S &= \{c\}
\end{aligned}$$

Moreover, by Turing completeness we are able to define a regular command r_w such that, for all $S \in C$ we have $\llbracket \mathsf{r}_w \rrbracket S = \emptyset$. As an example, for a single variable `x` these four commands are

$$\begin{aligned}
\mathsf{r}_w &\triangleq \texttt{while (true) \{ skip \}} \\
\mathsf{r}_{c?} &\triangleq \texttt{if (x == c) then skip else } \mathsf{r}_w \\
\mathsf{r}_{\neg c?} &\triangleq \texttt{if (x == c) then } \mathsf{r}_w \texttt{ else skip} \\
\mathsf{r}_{c} &\triangleq \texttt{x := c}
\end{aligned}$$

Since A is not trivial, there exists two set of stores P and R such that $P \subsetneq A(P)$ and $A(R) \subsetneq C$. These imply that there exists two stores $a \in A(P) \setminus P$ and $b \in C \setminus A(R)$. Since by monotonicity of A we get $A(\emptyset) \subseteq A(R)$ and we know $b \notin A(R)$, we conclude that $A(\{b\}) \neq A(\emptyset)$. Moreover, by monotonicity of A we also have $A(\emptyset) \subseteq A(\{b\})$, so that $A(\emptyset) \subsetneq A(\{b\})$. Given such a and b, let us consider the command r defined as

$$\mathsf{r} = (\mathsf{r}_{a?}; \mathsf{r}_b) \oplus (\mathsf{r}_{\neg a?}; \mathsf{r}_w)$$

for $\mathsf{r}_{a?}$, r_b, $\mathsf{r}_{\neg a?}$ and r_w as defined above. We now show that we cannot prove the triple $\vdash_A [P]\ \mathsf{r}_1; \mathsf{r}_w\ [\emptyset]$ in LCL$_A$ extended with (simplify) even though it is intensionally sound.

First, we verify soundness: $\llbracket \mathsf{r}; \mathsf{r}_w \rrbracket P = \llbracket \mathsf{r}_w \rrbracket (\llbracket \mathsf{r} \rrbracket P) = \emptyset$, so that $\emptyset \subseteq \llbracket \mathsf{r}; \mathsf{r}_w \rrbracket P$, and

$$\alpha(\llbracket \mathsf{r}; \mathsf{r}_w \rrbracket P) = \alpha(\emptyset) = \llbracket \mathsf{r}_w \rrbracket^{\sharp}_{A}(\llbracket \mathsf{r} \rrbracket^{\sharp}_{A} \alpha(P)) = \llbracket \mathsf{r}; \mathsf{r}_w \rrbracket^{\sharp}_{A} \alpha(P)$$

so the triple is intensionally sound.

However, we can't derive the triple because r is not locally complete on P in A, nor in any $A' \succeq A$ such that $A(\emptyset) = A'(\emptyset)$, that are the only domain we can simplify to using rule (simplify). The local incompleteness is a consequence of

$$\llbracket \mathsf{r} \rrbracket A(P) = \llbracket \mathsf{r}_{a?}; \mathsf{r}_b \rrbracket A(P) \cup \llbracket \mathsf{r}_{\neg a?}; \mathsf{r}_w \rrbracket A(P) = \llbracket \mathsf{r}_b \rrbracket \llbracket \mathsf{r}_{a?} \rrbracket A(P) \cup \emptyset = \llbracket \mathsf{r}_b \rrbracket \{a\} = \{b\}$$

where we used $a \in A(P)$. Using this we easily get $\neg\mathbb{C}^A_P(\llbracket \mathsf{r} \rrbracket)$:

$$A(\llbracket \mathsf{r} \rrbracket P) = A(\emptyset) \subsetneq A(\{b\}) = A(\llbracket \mathsf{r} \rrbracket A(P))$$

Formally, to derive the triple $\vdash_A [P]\ \mathsf{r}; \mathsf{r}_w\ [\emptyset]$, we can apply only three rules: (relax), (seq) and (simplify). By the arbitrariness of P we can assume without loss of generality that there is no $P' \subseteq P \subseteq A(P')$, and clearly any $Q' \supseteq \emptyset$ is no longer sound because $\emptyset = \llbracket \mathsf{r}; \mathsf{r}_w \rrbracket P$. So we can't apply (relax). Let us distinguish the other two cases.
If we apply (seq), we get

$$\frac{\vdash_A [P]\ \mathsf{r}\ [Q] \quad \vdash_A [Q]\ \mathsf{r}_w\ [\emptyset]}{\vdash_A [P]\ \mathsf{r}; \mathsf{r}_w\ [\emptyset]}\ (\mathsf{seq})$$

for some intermediate Q. However, any provable triple $\vdash_A [P]\ \mathsf{r}\ [Q]$ would imply by soundness (of LCL$_A$ extended with (simplify), Theorem 6.12) local completeness $\mathbb{C}^A_P(\llbracket \mathsf{r} \rrbracket)$, that is false. So we can't prove the triple starting with an application of (seq)
If we apply (simplify), we get

$$\frac{\vdash_A [P]\ \mathsf{r}; \mathsf{r}_w\ [\emptyset] \quad A' \succeq A \quad A'(\emptyset) = A(\emptyset)}{\vdash_A [P]\ \mathsf{r}; \mathsf{r}_w\ [\emptyset]}\ (\mathsf{simplify})$$

for some simplification A' of A. Note that we have $A'(\emptyset) \subsetneq A'(\{b\})$:

$$A'(\emptyset) = A(\emptyset) \subsetneq A(\{b\}) \subseteq A'(\{b\})$$

Now we could apply (relax) to change P with a P' satisfying $P' \subseteq P \subseteq A'(P')$. However, we have $a \notin P$, hence $a \notin P' \subseteq P$, and $a \in A(P) \subseteq A'(P) = A'(P')$. Now the only rule we can apply is (seq), and as before we need a triple $\vdash_{A'} [P']\ \mathsf{r}\ [Q]$ for some Q, that would imply $\mathbb{C}^{A'}_{P'}(\llbracket \mathsf{r} \rrbracket)$. But this is not the case, and can be shown with computations as before using that $a \in A'(P') \setminus P'$ to obtain $\llbracket \mathsf{r} \rrbracket P' = \emptyset$ and $\llbracket \mathsf{r} \rrbracket A'(P') = \{b\}$, and the already shown $A'(\emptyset) \subsetneq A'(\{b\})$. □

Proof of Theorem 6.18. The proof follows closely that for LCL (Theorem 6.2).
As for LCL$_A$, points (1) and (3) implies point (2):

$$\begin{aligned} \alpha(P) &\leq \alpha(\llbracket \overleftarrow{\mathsf{r}} \rrbracket Q) && [(1) \text{ and monotonicity of } \alpha] \\ &\leq \llbracket \overleftarrow{\mathsf{r}} \rrbracket^\sharp_A \alpha(Q) && [\text{soundness of } \llbracket \overleftarrow{\cdot} \rrbracket^\sharp_A] \\ &= \alpha(P) && [(3)] \end{aligned}$$

Therefore, we only have to prove (1) and (3).
The proof is by induction on the derivation tree of the provable triple $\vdash_A \langle\!\langle P \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q \rangle\!\rangle$.
Case (transfer)
For (1), we have trivially $\llbracket \overleftarrow{\mathsf{c}} \rrbracket Q \leq \llbracket \overleftarrow{\mathsf{c}} \rrbracket Q$. For (3), we have $\alpha(\llbracket \overleftarrow{\mathsf{c}} \rrbracket Q) = \alpha(\llbracket \overleftarrow{\mathsf{c}} \rrbracket \gamma\alpha Q) = \llbracket \overleftarrow{\mathsf{c}} \rrbracket^\sharp_A \alpha(Q)$ by $\mathbb{C}^A_Q(\llbracket \overleftarrow{\mathsf{c}} \rrbracket)$ and definition of $\llbracket \overleftarrow{\mathsf{c}} \rrbracket^\sharp_A$, respectively (see Figure 6.4).
Case (relax)
For (1) we have by inductive hypothesis $P' \leq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q'$, and together with the other hypotheses of the rules we get $P \leq P' \leq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q' \leq \llbracket \overleftarrow{\mathsf{r}} \rrbracket Q$.
For (3), we recall that $P \leq P' \leq A(P)$ implies $\alpha(P) = \alpha(P')$: one inequality holds by monotonicity of α, the other by the adjunctive property of a GC since $A = \gamma\alpha$. Then

$$\begin{aligned} \alpha(P) &= \alpha(P') && [\text{hp of the rule } P \leq P' \leq A(P)] \\ &= \llbracket \overleftarrow{\mathsf{r}} \rrbracket^\sharp_A \alpha(Q') && [\text{inductive hp (2) on } \vdash_A \langle\!\langle P' \rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q' \rangle\!\rangle] \\ &= \llbracket \overleftarrow{\mathsf{r}} \rrbracket^\sharp_A \alpha(Q) && [\text{hp of the rule } Q' \leq Q \leq A(Q')] \end{aligned}$$

Case (seq)
For (1) $P \leq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket R \leq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket(\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket Q) = \llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket Q$, where the inequalities follow from inductive hypotheses and monotonicity of $\llbracket\overleftarrow{\mathsf{r}_1}\rrbracket$.
For (3), we recall that $\llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket^A \leq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket^A\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket^A$.

$$\begin{aligned}
\alpha(P) &\leq \alpha(\llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket Q) && \text{[(1) and monotonicity of } \alpha\text{]}\\
&\leq \llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket^A\alpha(Q) && \text{[soundness of } \llbracket\overleftarrow{\mathsf{r}}\rrbracket^A\text{]}\\
&\leq \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket^A\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket^A\alpha(Q) && \text{[recalled above]}\\
&= \llbracket\mathsf{r}_1\rrbracket^A\alpha(R) && \text{[inductive hp]}\\
&= \alpha(P) && \text{[inductive hp]}
\end{aligned}$$

So all the lines are equal, in particular $\llbracket\overleftarrow{\mathsf{r}_1;\mathsf{r}_2}\rrbracket^A\alpha(Q) = \alpha(P)$.
Case (join)
For (1), by inductive hypotheses, $P_1 \leq \llbracket\mathsf{r}_1\rrbracket Q$ and $P_2 \leq \llbracket\mathsf{r}_2\rrbracket Q$. Hence, $P_1 \vee P_2 \leq \llbracket\mathsf{r}_1\rrbracket Q \vee \llbracket\mathsf{r}_2\rrbracket Q = \llbracket\mathsf{r}_1 \oplus \mathsf{r}_2\rrbracket Q$.
For (3), we observe that

$$\begin{aligned}
\llbracket\overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2}\rrbracket^A\alpha(Q) &= \alpha\llbracket\overleftarrow{\mathsf{r}_1 \oplus \mathsf{r}_2}\rrbracket\gamma\alpha(Q)\\
&= \alpha(\llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\gamma\alpha(Q) \vee \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\gamma\alpha(Q))\\
&= \alpha\llbracket\overleftarrow{\mathsf{r}_1}\rrbracket\gamma\alpha(Q) \vee \alpha\llbracket\overleftarrow{\mathsf{r}_2}\rrbracket\gamma\alpha(Q)\\
&= \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket^A\alpha(Q) \vee \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket^A\alpha(Q)
\end{aligned}$$

where we used additivity of α. Recalling that, by inductive hypotheses, $\alpha(P_1) = \llbracket\overleftarrow{\mathsf{r}_1}\rrbracket^A\alpha(Q)$ and $\alpha(P_2) = \llbracket\overleftarrow{\mathsf{r}_2}\rrbracket^A\alpha(Q)$, we get

$$\begin{aligned}
\alpha(P_1 \vee P_2) &= \alpha(P_1) \vee \alpha(P_2) && [\alpha \text{ is additive}]\\
&= \llbracket\mathsf{r}_1\rrbracket^A\alpha(Q) \vee \llbracket\mathsf{r}_2\rrbracket^A\alpha(Q) && \text{[inductive hypotheses]}\\
&= (\llbracket\mathsf{r}_1 \oplus \mathsf{r}_2\rrbracket^A)\alpha(Q) && \text{[observation above]}
\end{aligned}$$

Case (rec)
We first observe that

$$\begin{aligned}
\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q &\leq Q \vee \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q\\
&= \llbracket\overleftarrow{\mathsf{r}}\rrbracket^0 Q \vee \bigvee_{n \geq 0}\llbracket\overleftarrow{\mathsf{r}}\rrbracket^n\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q && \text{[Lemma 5.1]}\\
&= \bigvee_{n \geq 0}\llbracket\overleftarrow{\mathsf{r}}\rrbracket^n Q\\
&= \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q && \text{[Lemma 5.1]}
\end{aligned}$$

We can then prove (1) by

$$\begin{aligned}
P &\leq \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket(R \vee Q) && \text{[inductive hp (1) on } \vdash_A \langle\!\langle P\rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle R \vee Q\rangle\!\rangle\text{]}\\
&\leq \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket(\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q \vee Q) && \text{[inductive hp (1) on } \vdash_Q \langle\!\langle R\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle\text{]}\\
&= \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket\llbracket\overleftarrow{\mathsf{r}}\rrbracket Q \vee \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q && \text{[additivity of } \llbracket\overleftarrow{\cdot}\rrbracket\text{]}\\
&\leq \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q \vee \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q && \text{[observation above]}\\
&= \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q
\end{aligned}$$

For (3)

$$\begin{aligned}
\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A\alpha(Q) &\leq_A \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A(Q \vee R) \\
&= \alpha(P) && [\text{inductive hp (2) on } \vdash_A \langle\!\langle P\rangle\!\rangle\ \mathsf{r}^\star\ \langle\!\langle R \vee Q\rangle\!\rangle] \\
&\leq_A \alpha(\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q) && [\text{condition (1) shown above}] \\
&\leq_A \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A\alpha(Q) && [\text{soundness of } \llbracket\overleftarrow{\cdot}\rrbracket^\sharp_A]
\end{aligned}$$

Case (iterate)
For (1)

$$\begin{aligned}
P \vee Q &\leq \llbracket\overleftarrow{\mathsf{r}}\rrbracket Q \vee Q && [\text{inductive hp (1) on } \vdash_A \langle\!\langle P\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle] \\
&= \llbracket\overleftarrow{\mathsf{r}}\rrbracket^1 Q \vee \llbracket\overleftarrow{\mathsf{r}}\rrbracket^0 Q \\
&\leq \bigvee_{n\geq 0} \llbracket\overleftarrow{\mathsf{r}}\rrbracket^n Q \\
&= \llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket Q && [\text{Lemma 5.1}]
\end{aligned}$$

For (3), we first prove by induction that $(\llbracket\overleftarrow{\mathsf{r}}\rrbracket^\sharp_A)^n\alpha(Q) \leq_A \alpha(Q)$. Recall that, by the adjunctive property of a Galois connection, $P \leq A(Q)$ iff $\alpha(P) \leq_A \alpha(Q)$. The base case $n = 0$ is trivial because $(\llbracket\overleftarrow{\mathsf{r}}\rrbracket^\sharp_A)^0$ is the identity. Suppose it holds for some n: then

$$\begin{aligned}
(\llbracket\overleftarrow{\mathsf{r}}\rrbracket^\sharp_A)^{n+1}\alpha(Q) &= \llbracket\overleftarrow{\mathsf{r}}\rrbracket^\sharp_A(\llbracket\overleftarrow{\mathsf{r}}\rrbracket^\sharp_A)^n\alpha(Q) \\
&\leq_A \llbracket\overleftarrow{\mathsf{r}}\rrbracket^\sharp_A\alpha(Q) && [\text{inductive hp}] \\
&= \alpha(P) && [\text{inductive hp (2) on } \vdash_A \langle\!\langle P\rangle\!\rangle\ \mathsf{r}\ \langle\!\langle Q\rangle\!\rangle] \\
&\leq_A \alpha(Q) && [\text{hp of the rule } P \leq A(Q)]
\end{aligned}$$

From this, we observe that

$$\alpha(Q) = (\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A)^0\alpha(Q) \leq_A \bigvee_{n\geq 0}(\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A)^n\alpha(Q) \leq_A \bigvee_{n\geq 0}\alpha(Q) = \alpha(Q)$$

and therefore $\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A\alpha(Q) = \alpha(Q)$ by Lemma 5.1.

We then conclude the proof of (2) with

$$\begin{aligned}
\llbracket\overleftarrow{\mathsf{r}^\star}\rrbracket^\sharp_A\alpha(Q) &= \alpha(Q) \\
&= \alpha(P) \vee_A \alpha(Q) && [\text{hp of the rule } P \leq A(Q)] \\
&= \alpha(P \vee Q) && [\text{additivity of } \alpha]
\end{aligned}$$

□

Proof of Proposition 6.20.

Write the proof or decide I don't care

□

$$\dfrac{\dfrac{\mathbb{C}^{\mathrm{Oct}}_{x+n=\bar{k}}(\llbracket \overleftarrow{\texttt{n > 0?}} \rrbracket)}{\vdash_{\mathrm{Oct}} \langle\!\langle T_{2M} \rangle\!\rangle\ \texttt{n > 0?}\ \langle\!\langle x+n=\bar{k} \rangle\!\rangle} \quad \dfrac{\mathbb{C}^{\mathrm{Oct}}_{x=\bar{k}}(\llbracket \overleftarrow{\texttt{x := x + n}} \rrbracket)}{\vdash_{\mathrm{Oct}} \langle\!\langle x+n=\bar{k} \rangle\!\rangle\ \texttt{x := x + n}\ \langle\!\langle x=\bar{k} \rangle\!\rangle} \quad \dfrac{\mathbb{C}^{\mathrm{Oct}}_{R_{2M}}(\llbracket \overleftarrow{\texttt{n := nondet()}} \rrbracket)}{\vdash_{\mathrm{Oct}} \langle\!\langle x=\bar{k} \rangle\!\rangle\ \texttt{n := nondet()}\ \langle\!\langle R_{2M} \rangle\!\rangle}}{\vdash_{\mathrm{Oct}} \langle\!\langle T_{2M} \rangle\!\rangle\ \mathsf{r}_w\ \langle\!\langle R_{2M} \rangle\!\rangle}\ (\mathsf{seq})$$

$$\dfrac{\dfrac{\mathbb{C}^{\mathrm{Oct}}_{x+n\leq\bar{k}}(\llbracket \overleftarrow{\texttt{n > 0?}} \rrbracket)}{\vdash_{\mathrm{Oct}} \langle\!\langle S_{2M} \rangle\!\rangle\ \texttt{n > 0?}\ \langle\!\langle x+n\leq\bar{k} \rangle\!\rangle} \quad \dfrac{\mathbb{C}^{\mathrm{Oct}}_{x\leq\bar{k}}(\llbracket \overleftarrow{\texttt{x := x + n}} \rrbracket)}{\vdash_{\mathrm{Oct}} \langle\!\langle x+n\leq\bar{k} \rangle\!\rangle\ \texttt{x := x + n}\ \langle\!\langle x\leq\bar{k} \rangle\!\rangle} \quad \dfrac{\mathbb{C}^{\mathrm{Oct}}_{T_{2M}\vee R_{2M}}(\llbracket \overleftarrow{\texttt{n := nondet()}} \rrbracket)}{\vdash_{\mathrm{Oct}} \langle\!\langle x<\bar{k} \vee x=\bar{k} \rangle\!\rangle\ \texttt{n := nondet()}\ \langle\!\langle T_{2M}\vee R_{2M} \rangle\!\rangle}}{\vdash_{\mathrm{Oct}} \langle\!\langle S_{2M} \rangle\!\rangle\ \mathsf{r}_w\ \langle\!\langle T_{2M}\vee R_{2M} \rangle\!\rangle}\ (\mathsf{seq})$$

$$\dfrac{\dfrac{S_{2M} \leq \mathrm{Oct}(T_{2M}\vee R_{2M}) \quad \dfrac{\dots}{\vdash_{\mathrm{Oct}} \langle\!\langle S_{2M} \rangle\!\rangle\ \mathsf{r}_w\ \langle\!\langle T_{2M}\vee R_{2M} \rangle\!\rangle}}{\vdash_{\mathrm{Oct}} \langle\!\langle S_{2M}\vee R_{2M} \rangle\!\rangle\ \mathsf{r}^\star_w\ \langle\!\langle T_{2M}\vee R_{2M} \rangle\!\rangle}\ (\mathsf{iterate}) \quad \dfrac{\dots}{\vdash_{\mathrm{Oct}} \langle\!\langle T_{2M} \rangle\!\rangle\ \mathsf{r}_w\ \langle\!\langle R_{2M} \rangle\!\rangle}}{\vdash_{\mathrm{Oct}} \langle\!\langle S_{2M}\vee R_{2M} \rangle\!\rangle\ \mathsf{r}^\star_w\ \langle\!\langle R_{2M} \rangle\!\rangle}\ (\mathsf{rec})$$

Figure D.3: Derivation of the CLCL triple $\vdash_{\mathrm{Oct}} \langle\!\langle (n > 0 \wedge x + n \leq 2000000) \vee R_{2M} \rangle\!\rangle\ \mathsf{r}^\star_w\ \langle\!\langle R_{2M} \rangle\!\rangle$. For brevity, we omit rule name (transfer), we let $\bar{k} \triangleq 2000000$, we recall that $R_{2M} \triangleq (x = \bar{k} \wedge n \leq 0)$, $T_{2M} \triangleq (x + n = \bar{k} \wedge n > 0)$ and we define $S_{2M} \triangleq (x + n \leq \bar{k} \wedge n > 0)$.

Appendix E

Appendix

This appendix contains technical details of proofs and examples for Chapter 7.

From here

Todo list