Cartesian Abstraction and Verification of Multithreaded Programs

Dissertation
zur
Erlangung des Doktorgrades
der Technischen Fakultät
der Albert-Ludwigs-Universität
Freiburg im Breisgau

vorgelegt von
Alexander Malkis

Dekan: Prof. Dr. Hans Zappe
Referenten:
Prof. Dr. Andreas Podelski (Doktorvater und Erstgutachter),
Prof. Dr. Ernst-Rüdiger Olderog (Zweitgutachter)
Datum des Promotionsvortrages: 10.02.2010
Datum der Aushändigung der Urkunde: 09.07.2010

Meinen Eltern gewidmet.

Zusammenfassung

Kurzdarstellung

Ein Verifikationsalgorithmus für nebenläufige Programme, der in der Anzahl der Threads skaliert, ist zwangsläufig unvollständig. Für vollständige Algorithmen kann man höchstens ein polynomielles Verhalten auf einer praktisch interessanten Klasse erwarten. In dieser Arbeit geben wir einen solchen Algorithmus an. Der Algorithmus berechnet iterativ eine gegenbeispielgesteuerte Verfeinerung einer bekannten thread-modularen Methode. Der Algorithmus ist beweisbar polynomiell auf der Klasse der Programme mit Locks. Experimente zeigen, dass er mit der Klasse der Programme mit Locks auch in der Praxis gut auskommt.

Einführung

Geehrter Leser!

Die vorliegende Arbeit widmet sich den Beweisen von Programmkorrektheit. Effiziente Beweismethoden sind nicht auf alle Programme anwendbar. Die Frage ist, auf welchen Bereich die effizienten Methoden anwendbar sind und wie sich dieser Bereich auf weitere Programme ausdehnen lässt. Die Dissertation charakterisiert die Klasse der Programme, bei denen die genannten Methoden greifen, und verbessert die Methoden so, dass sie auf eine größere Anzahl von Programmen anwendbar sind.

Konkret betrachtet die Dissertation thread-modulare Korrektheitsbeweise nebenläufiger Programme. Ein nebenläufiges Programm besteht aus einer festen Anzahl von Threads. Threads sind sequenzielle Programme, die über einen gemeinsamen Speicher kommunizieren; ein Schritt des nebenläufigen Programms ist ein Schritt eines der Threads. Thread-modulare Verfahren bilden eine besondere Klasse der effizienten Verfahren zum Erstellen der Korrektheitsbeweise. Jedoch schlagen die thread-modularen Beweisversuche vieler Verifikationsaufgaben (also der Programme zusammen mit ihren Eigenschaften) aus unterschiedlichen Gründen fehl. Die Dissertation erklärt die allgemeine Quelle solcher Fehlschläge und zeigt, wie man existierende thread-modulare Verfahren auf mehr Verifikationsaufgaben ausdehnen kann.

Die Dissertation konzentriert sich auf die Sicherheitseigenschaften: benutzerdefinierte Belegungen von Variablen sollen in keinem Programmablauf vorkommen. Die Sicherheit eines nebenläufigen Programms ist ein PSPACE-hartes Problem in der Anzahl der Threads. Thread-modulare Verfahren betrachten Threads getrennt voneinander; sie haben polynomielle Laufzeit, können aber

nicht alle Verifikationsaufgaben lösen. Gibt es eine Möglichkeit, die Verifikationsaufgaben zu beschreiben, die thread-modulare Beweise zulassen? Die Dissertation charakterisiert die Klasse der Verifikationsaufgaben mit thread-modularen Beweisen. Wenn eine Verifikationsaufgabe keinen thread-modularen Beweis zulässt, wie kann man einen thread-modularen Beweisversuch für diese ausgewählte Verifikationsaufgabe gewissermaßen "reparieren"? Lässt sich ein thread-modularer Beweisversuch parametrisieren? Ist der damit verbundene zusätzliche Laufzeitaufwand vertretbar, so würde das "Reparieren" erlauben, effiziente thread-modulare Beweismethoden auf mehr Verifikationsaufgaben auszudehnen. Die Dissertation löst dieses Problem, indem sie zeigt, wie man thread-modulare Beweise parametrisieren kann. Wenn Parametrisierung möglich ist, ist es möglich, einen adäquaten Parameter automatisch zu finden? Die Dissertation löst auch diese Aufgabe, indem sie den Algorithmus angibt, der den Parameter findet und dabei die Zeit gemäß der Komplexität des Beweises verbraucht: harte Beweise brauchen mehr Zeit als einfache.

Um die Unvollständigkeit der thread-modularen Beweisführung zu verstehen, wendet sich die Dissertation der Theorie der abstrakten Interpretation zu, die einen formellen Rahmen für die Beschreibung von Programmanalysen bildet. Zum Beispiel könnte man ein sequenzielles Programm mit n Variablen analysieren, indem man für jede Variable die Menge der Werte speichert, die diese Variable annehmen kann; dabei leitet die Analyse schrittweise neue Variablenwerte her und approximiert in jedem Schritt die Menge der Belegungen durch ein kartesisches Produkt. Etwas Ähnliches lässt sich mit einem nebenläufigen Programm aus n Threads durchspielen. Stellt ein n-Tupel die lokalen Zustände der Threads dar, wobei jede Komponente einen lokalen Zustand eines einzelnen Threads darstellt, so kann man die Menge der n-Tupel durch ein kartesisches Produkt approximieren. Vereinfacht gesprochen leitet die Analyse des nebenläufigen Programms iterativ neue lokale Zustände her und approximiert jedes Mal die Menge der Tupel durch kartesische Produkte. Dieses Verfahren, welches man nebenläufige kartesische Programmanalyse nennt, verbraucht Platz proportional zur Anzahl der Threads. Interessanterweise stellt sich heraus, dass die Präzision dieses Verfahrens äquivalent zu der der thread-modularen Beweisführung ist. Kann also eine thread-modulare Beweisführung eine Eigenschaft nicht zeigen, könnte man genauso gut die nebenläufige kartesische Programmanalyse durchführen und ihre Berechnung betrachten. Der früheste Punkt in der Berechnung, ab dem sich eine zu grobe Überapproximation einstellt, kann exakt beschrieben werden. Darüber hinaus kann diese Information dazu benutzt werden, den Beweis zu "reparieren", und zugleich einen Parameter für die thread-modulare Beweisführung zu finden. Am oben genannten Punkt weist eine Tupelmenge eine zu grobe kartesische Approximation auf; die Approximation einer kleineren Menge wäre dagegen unproblematisch. In diesem Fall lässt sich die kleinere Menge anstelle der größeren approximieren und die Differenz zwischen der großen und der kleinen Menge später hinzufügen. Diese Differenz wird eine Ausnahmemenge genannt; sie ist der gesuchte Parameter. Sie wird vor der Approximation abgezogen und nach der Approximation hinzugefügt. Die Invariante (d.h. eine Beschreibung der Belegungen der Programmvariablen, welche für jeden Zeitpunkt in jedem Ablauf zutrifft), welche die kartesische Approximation herleitet, besteht aus kartesischen Produkten und Ausnahmemengen. Die Dissertation erläutert das notwendige Hintergrundwissen zu den kartesischen Produkten und der Programmanalyse und erklärt Beweisreparaturen mit

Ausnahmemengen sowie die thread-modulare Beweisführung. Anschließend wird aus diesen Komponenten ein Algorithmus für Korrektheitsbeweise nebenläufiger Programme konstruiert, der die thread-modulare Beweisführung erweitert.

Bei der Untersuchung der oben genannten Fragen tauchen Probleme auf, die außerhalb des betrachteten Bereichs liegen. Diese werden am Ende der Dissertation kurz aufgeführt.

Einige technische Details der Dissertation sind zum Verständnis nicht zwingend notwendig und können bei einer ersten Lektüre übersprungen werden. Die entsprechenden Abschnitte sind mit einem Stern (*) markiert.

Contents

Introduction

Dear reader!

The thesis in front of you deals with proving program correctness. Efficient methods for that are, however, not applicable to all programs. It would be nice to understand the current area of applicability of these methods and to extend this area to more programs. The thesis characterizes the programs on which these efficient methods can be applied and improves the methods so that they can be applied to more programs.

In particular, the thesis deals with thread-modular correctness proofs of multithreaded programs. Threads are sequential programs communicating via shared memory, a multithreaded program consists of a fixed number of threads, a step of the multithreaded program is a step of some of the threads. Thread-modular techniques are a particular class of effective techniques for proving correctness. However, thread-modular proof attempts of many verification tasks (i.e. programs together with their properties) fail for different reasons. The thesis explains the general source of such failures and shows how to extend existing thread-modular techniques to more verification tasks.

The thesis concentrates on proving safety, i.e. proving that user-given variable valuations never occur in computations. Proving safety of a multithreaded program is a PSPACE-hard problem in the number of threads. Thread-modular techniques reason about each thread in separation; they take polynomial time but cannot deal with all verification tasks. Is there a way to describe verification tasks that admit thread-modular proofs? The thesis precisely characterizes the class of verification tasks with thread-modular proofs. If a verification task doesn't admit a thread-modular proof, how could one "repair" a thread-modular proof attempt for that particular task? Is it possible to parameterize a thread-modular proof? Assuming a reasonable time penalty, repairing would allow applying efficient thread-modular methods to more tasks. The thesis solves this problem, showing how to parameterize thread-modular proofs. If parameterizing is possible, is it possible to find the parameter automatically? The thesis also solves this problem by giving an algorithm for finding the parameter, smoothly trading off time for proof complexity: complex parameterizations require more time than simple ones.

To understand the incompleteness of thread-modular reasoning, the thesis turns to the theory of abstract interpretation, which is a framework for describing program analyses. For instance, one may analyze a sequential program on n variables by storing for each variable a set of values that it can take; the analysis iteratively derives new values of the variables and at each step approximates the set of all valuations by a Cartesian product. A similar game can be played on an n-threaded program. If an n-tuple represents local states of

the threads, where each component represents a local state of a distinct thread, one can approximate a set of n-tuples by a Cartesian product. Simplifying, an analysis of the multithreaded program iteratively derives new local states of the threads and each time approximates sets of tuples of local states by Cartesian products. This game, called multithreaded Cartesian program analysis, needs linear space in the number of threads. Interestingly, its precision turns out to be equivalent to that of thread-modular reasoning! So if thread-modular techniques fail to prove a property, one might as well look at the computation of the multithreaded Cartesian program analysis; there the earliest point at which a too coarse approximation happens is exactly identifiable. Moreover, this information can be used to "repair" the proof, at the same time finding a parameter for the thread-modular method. Assume that some set of tuples has a too coarse Cartesian approximation but the approximation of a smaller set would make no problem. Then, instead of approximating the large set, one approximates the smaller set and adds the difference between the larger and the smaller set back. This difference is called the exception set, which is the wanted parameter. It gets subtracted before the approximation and gets added back after it. The invariant (i.e. a description of valuations of variables that holds for all time points in any computation) that multithreaded Cartesian program analysis derives consists of Cartesian products and exception sets. The thesis explains the necessary facts about Cartesian products, program analysis, repairing proofs via exceptions, thread-modular reasoning and combines these ingredients into an algorithm for verifying multithreaded programs that extends thread-modular reasoning.

The thesis is concluded by a list of open problems. Thesis parts marked by a star (*) can be skipped during first reading.

Chapter 1

Thread-Modular Reasoning

The current chapter explains and connects three different approaches to verifying multithreaded programs. The three different approaches are Cartesian abstraction, thread-modular model-checking and the Owicki-Gries proof system. These approaches come from different worlds, they exist for different purposes and use different notation. The most striking result of the chapter is the precision equivalence: the three classes of properties that are verifiable by the respective approaches are equal.

1.1 Multithreaded Programs

In the following we ignore the internal structure of a thread, viewing each thread just as a transition system. Our work aims at reducing the complexity in the number of threads and not in any other parameter like the number of variables.

Definition 1.1.1. A *multithreaded program* is a tuple

$$(\mathrm{Glob}, \mathrm{Loc}, (\to_i)_{i=1}^n, \mathrm{init}),$$

where Glob and Loc are any sets, each $\to_i$ is a subset of $(\mathrm{Glob} \times \mathrm{Loc})^2$ (for $1 \leq i \leq n$) and $\mathrm{init} \subseteq \mathrm{Glob} \times \mathrm{Loc}^n$.

The components of the multithreaded program mean the following:

- Loc, the set of *local states*, contains valuations of local variables (including the program counter) of any thread (without loss of generality let all threads have equal sets of local states; if the set of valuations of local variables of thread i is Loc_i, take, e.g., $\mathrm{Loc} = \bigcup_{i=1}^n \mathrm{Loc}_i$);
- Glob, the set of *shared states*, contains valuations of shared variables;
- the elements of $\mathrm{States} = \mathrm{Glob} \times \mathrm{Loc}^n$ are called *program states*, the elements of $\mathrm{Glob} \times \mathrm{Loc}$ are called *thread states*, the projection on the shared component and the i^{th} local component is the map

$$\pi_{0,i} : 2^{\mathrm{States}} \to 2^{\mathrm{Glob} \times \mathrm{Loc}}, \quad S \mapsto \{(g, l_i) \mid (g, l) \in S\},$$

  where l_i is the i^{th} component of the vector l;
- the relation $\to_i$ is the transition relation of the i^{th} thread ($1 \leq i \leq n$);
- init is the set of *initial states*.

The program is called *finite-state* if Glob and Loc are finite, in this case its *size* is $|\text{Glob}| + |\text{Loc}| + \sum_{i=1}^{n} |{\to_i}| + |\text{init}|$.
The program is equipped with the interleaving semantics. This means that if a thread makes a step, then it may change its own local variables and the global variables but may not change the local variables of another thread; a step of the whole program is a step of some of the threads. The *post* operator maps a set of states to the set of their successors:

$$\begin{aligned} \text{post} :\ & 2^{\text{States}} \to 2^{\text{States}}, \\ S \ \mapsto\ & \{(g', l') \mid \exists\, (g, l) \in S,\, i \in \mathbb{N}_n : (g, l_i) \to_i (g', l'_i) \\ & \qquad\qquad \text{and } \forall\, j \neq i :\ l_j = l'_j\}\,, \end{aligned}$$

where $\mathbb{N}_n$ is the set of first n positive integers. We are interested in proving safety properties of multithreaded programs. Each safety property can be encoded as a non-reachability property. So we want to show that any computation that starts in an initial state stays within the set of safe states, formally:

$$\bigcup_{k \geq 0} \text{post}^k(\text{init}) \subseteq \text{safe}\,.$$

1.2 Inference rules*

We will describe thread-modular verification by means of inference rules. Now we give semantics to such descriptions. A reader who feels comfortable with inference rules may skip this part during first reading.

Sometimes sets are defined inductively by means of inference rules like

$$\frac{a \in X_1 \qquad b \in X_2}{b \in X_1 \qquad c \in X_3} \qquad \frac{c \in X_3 \qquad d \in X_3}{c \in X_2} \qquad \ldots \tag{1.1}$$

where X_1, X_2 and X_3 are set variables and a, b, c, d are constants. A rule scheme might be given which represents a collection of many (perhaps infinitely many) rules:

$$\frac{x \in X \qquad y \in Y}{xy \in Z}\,.$$

The above rule says that Z should contain the products of elements of X and Y.

What do these rules define, strictly speaking? Intuitively speaking, when each rule is interpreted as a constraint, the rules define the least set or set tuple that satisfies all the constraints. Now we'll formalize this notion.

To start formalizing, let's assume that we have n sets D_i ($i \in \mathbb{N}_n$) and we would like to define n subsets $X_i \subseteq D_i$ ($i \in \mathbb{N}_n$). A *rule* is a triple $((A_i)_{i=1}^n, k, x)$ where $A_i \subseteq D_i$ ($i \in \mathbb{N}_n$), $k \in \mathbb{N}_n$ and $x \in D_k$. For example, the two rules from (1.1) are formalized as triples $((\{a\}, \{b\}, \emptyset), 1, b)$, $((\{a\}, \{b\}, \emptyset), 3, c)$ and $((\emptyset, \emptyset, \{c, d\}), 2, c)$.

Consider the lattice $L := \prod_{i=1}^n \mathfrak{P}(D_i)$ equipped with the product order $\sqsubseteq = \{((X_i)_{i=1}^n, (X'_i)_{i=1}^n) \in L^2 \mid \forall\, i \in \mathbb{N}_n : X_i \subseteq X'_i\}$ and the least element $\bot = (\emptyset)_{i=1}^n$.

Definition 1.2.1. A *postfixpoint* of a set of rules R is a tuple $X = (X_i)_{i=1}^n$ such that for any $(A, i, x) \in R$ where $A \sqsubseteq X$ we have $x \in X_i$.

A *prefixpoint* of a set of rules R is a tuple $X = (X_i)_{i=1}^n$ such that for any $i \in \mathbb{N}_n$ and any $x \in X_i$ there is some $A \sqsubseteq X$ such that $(A, i, x) \in R$.

A *fixpoint* of R is a pre- and postfixpoint of R.

The *derivation operator* of R is

$$\begin{array}{rccl} \mathbb{D}_R : & L & \to & L \\ & X & \mapsto & (\{x \in D_i \mid \exists\, A \sqsubseteq X : (A, i, x) \in R\})_{i=1}^n \end{array}$$

Proposition 1.2.2. The prefixpoints (resp. postfixpoints, fixpoints) of a set of rules are exactly the prefixpoints (resp. postfixpoints, fixpoints) of its derivation operator.

Proof. Follows from the definitions. □

The derivation operator of a set of rules is monotone, thus the Tarski's fixpoint theorem (see Thm. 1 in [16]) implies the existence of the least fixpoint. The usual interpretation of a set of rules is that they define the least fixpoint of $\mathbb{D}_R$.

Theorem 1.2.3. If each rule has only finitely many premises, ω iterations suffice to get the least fixpoint. Formally:

$$(\forall\, ((A_i)_{i=1}^n, i, x) \in R, i \in \mathbb{N}_n : A_i \text{ is finite}) \quad \Rightarrow \quad \operatorname{lfp} R \;=\; \bigsqcup_{j \in \mathbb{N}_0} \mathbb{D}_R^j(\bot)\,.$$

where $\mathbb{N}_0$ is the set of nonnegative integers.

Idea. From high-level view, $\mathbb{D}_R$ is Scott-continuous (i.e. preserves suprema of directed subsets), thus Kleene's fixpoint theorem can be applied. We give a direct proof here.

Proof. To show that it's a postfixpoint, take any rule (A, k, x) of R where $A = (A_i)_{i=1}^n$ and let $A \sqsubseteq \bigsqcup_{j \in \mathbb{N}_0} \mathbb{D}_R^j(\bot)$. Since for each $i \in \mathbb{N}_n$ the set A_i from the rule is finite, the elements of A_i are in the union of finitely many iterates: $\forall\, i \in \mathbb{N}_n\ \exists\, m(i) \in \mathbb{N}_0 : A_i \subseteq (\bigsqcup_{j=0}^{m(i)} \mathbb{D}_R^j(\bot))_i$. Thus, $A \sqsubseteq \bigsqcup_{j=0}^{\max\{m(i) \mid i \in \mathbb{N}_n\}} \mathbb{D}_R^j(\bot)$. The derivation operator is monotone, so are its finite powers. Thus $\bot \sqsubseteq \mathbb{D}_R(\bot)$ implies $\mathbb{D}_R^{j+1}(\bot) \sqsupseteq \mathbb{D}_R^j(\bot)$ for all $j \in \mathbb{N}_0$. Especially $\bigsqcup_{j=0}^{\max\{m(i) \mid i \in \mathbb{N}_n\}} \mathbb{D}_R^j(\bot) \sqsubseteq \mathbb{D}_R^{\max\{m(i) \mid i \in \mathbb{N}_n\}}(\bot)$. So $x \in (\mathbb{D}_R^{\max\{m(i) \mid i \in \mathbb{N}_n\}+1}(\bot))_k$.

To show that this postfixpoint is the least one, it suffices to show that it is less than or equal to any postfixpoint. So let $C = (C_i)_{i=1}^n$ be any postfixpoint of R. We show by induction on j that $\forall\, j \in \mathbb{N}_0 : \mathbb{D}_R^j(\bot) \sqsubseteq C$.

For $j = 0$ we have $\mathbb{D}_R^0(\bot) = \bot \sqsubseteq C$.

Assume that the statement is proven for some $j \in \mathbb{N}_0$, i.e. $\mathbb{D}_R^j(\bot) \sqsubseteq C$. Take any $i \in \mathbb{N}_n$; it suffices to show that $\{x \in D_i \mid \exists\, A \sqsubseteq \mathbb{D}_R^j(\bot) : (A, i, x) \in R\} \subseteq C_i$. So let $x \in D_i$ such that there is some $A \sqsubseteq \mathbb{D}_R^j(\bot)$ with $(A, i, x) \in R$. Then $A \sqsubseteq C$. Since C is closed under the rule (A, i, x), we have $x \in C_i$.

So $\bigsqcup_{j \in \mathbb{N}_0} \mathbb{D}_R^j(\bot)$ is the least postfixpoint of the derivation operator. By Tarski's fixpoint theorem (Thm. 1 in [16]) it's also the least fixpoint. □

Thus each element of each set X_i ($i \in \mathbb{N}_n$) of the least fixpoint has a finite derivation tree.

1.3 Thread-modular model checking

Now we are going to present an algorithm that accumulates for each thread those thread states that can be potentially reachable. This algorithm was discovered by Cormac Flanagan and Shaz Qadeer at Microsoft Research in Redmond, and presented at the SPIN 2003 workshop [5].

The algorithm computes sets $\mathcal{R}_i \subseteq \text{Glob} \times \text{Loc}_\text{i}$ and $\mathcal{G}_i \subseteq \text{Glob} \times \text{Glob}$ ($i \in \mathbb{N}_n$) defined by the least fixed point of the following inference rules:

$$\text{INIT}\ \frac{}{\text{init}_i \subseteq \mathcal{R}_i} \qquad \text{STEP}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,l) \to_i (g',l')}{(g',l') \in \mathcal{R}_i \qquad (g,g') \in \mathcal{G}_i}$$

$$\text{ENV}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,g') \in \mathcal{G}_j}{(g',l) \in \mathcal{R}_i}\ i \neq j$$

Here, $\text{init}_i = \{(g,l_i) \mid (g,l) \in \text{init}\}$.

We call this system of inference rules FQ. The rules work as follows. The STEP rule discovers successors of a state of a thread that result due to a step of the same thread. Further, it stores the information about how the step changed the globals in the sets $\mathcal{G}_i$. The ENV rule uses this information to discover successors of a state of a thread that result due to communication between threads via globals. In the least fixpoint, the sets $\mathcal{R}_i$ ($i \in \mathbb{N}_n$) contain all those thread states that occur in computations (and may contain more thread states). For finite-state programs the least fixpoint is computable by a finite number of applications of the rules.

1.4 Represented program states

Now we show what program states are implicitly discovered by the Flanagan-Qadeer algorithm.

The inference rules of the Flanagan-Qadeer algorithm define the sets $\mathcal{R}_i$ ($i \in \mathbb{N}_n$) of "discovered" thread states. These sets represent those program states, whose globals and locals of the first thread are in $\mathcal{R}_1$, globals and locals of the second thread are in $\mathcal{R}_2$, and so on:

$$R^\# \ := \ \{(g,l) \mid \forall\, i \in \mathbb{N}_n : (g,l_i) \in \mathcal{R}_i\}\,.$$

If upon termination no error state is in $R^\#$, the algorithm says "the program is safe", otherwise it says "don't know".

The set $R^\#$ is usually not explicitly constructed. Usually the error condition is given by assertions in the program code for the threads. So a program state is erroneous if one of its corresponding thread states fails on an assertion. Thus to check that a program state $(g,l) \in R^\#$ is not an error state, it suffices to check that all its corresponding thread states (g,l_i) don't fail on assertions ($i \in \mathbb{N}_n$).

Here is a small example. The program below has one global variable g that can take values 0 or 1, each of two threads has a single local variable pc with values in $\{A,B,C,D\}$, representing the program counter.

$$\begin{array}{c} \text{Initially } g = 0 \\ \begin{array}{lll} A: & g := 0; & \| \quad C: \quad g := 1; \\ B: & & \| \quad D: \end{array} \end{array}$$

The algorithm discovers the following thread states:

$$\mathcal{R}_1 = \{(0, A), (0, B), (1, A), (1, B)\},$$
$$\mathcal{R}_2 = \{(0, C), (0, D), (1, D)\}.$$

where the pair $(0, A)$, for instance, is a shorthand for the pair of two maps $([g \mapsto 0], [pc \mapsto A])$. These two sets represent the set of program states

$$\{(g, l_1, l_2) \mid (g, l_1) \in \mathcal{R}_1 \text{ and } (g, l_2) \in \mathcal{R}_2\} =$$
$$\{(0, A, C), (0, A, D), (0, B, C), (0, B, D), (1, A, D), (1, B, D)\},$$

where the triple $(0, A, C)$, for instance, is a shorthand for the triple of maps $([g \mapsto 0], [pc \mapsto A], [pc \mapsto C])$.

1.5 Multithreaded Cartesian abstraction

What happens in the Flanagan-Qadeer algorithm? How precise is it? To answer this question we turn to the theory of abstract interpretation.

Let L be any set and $n \in \mathbb{N}^+$ where $\mathbb{N}^+$ denotes the set of all positive integers. Let $D = \mathfrak{P}(L^n)$ be the power set of the set of n-tuples over L and $D^\# = (\mathfrak{P}(L))^n$ be the set of n-tuples over subsets of L. We view $(D, \subseteq)$ as the concrete lattice and $(D^\#, \sqsubseteq)$ as the abstract lattice where $\sqsubseteq$ is the product order. The pair of maps

$$\begin{aligned} \alpha_c &: D \to D^\#, \qquad S \mapsto (\pi_i(S))_{i=1}^n, \\ \gamma_c &: D^\# \to D, \qquad (T_i)_{i=1}^n \mapsto \textstyle\prod_{i=1}^n T_i, \end{aligned} \tag{1.2}$$

where π_i is the projection to the i^{th} component, is a Galois connection (i.e. $\forall\, S \in D,\ T \in D^\# :\ \alpha_c(S) \sqsubseteq T \Leftrightarrow S \subseteq \gamma_c(T)$), called the Cartesian Galois connection. Variants are discussed by Patrick and Radhia Cousot in [3].

In order to relate the abstraction to the analyzed program, we need to know the meaning of L and n. For a sequential program a natural choice for L is the set of values of all the variables including the program counter, while n is the number of variables. Since a concurrent program is a special case of a nondeterministic sequential program, one might still choose the same granularity as before, namely separate the variables from each other. But it turns out that the resulting analysis would be very imprecise. A more natural and precise choice for L is the state space of a single thread, i.e. a valuation of all the variables including the program counter of a thread. However, there are some shared variables, i.e. variables that are accessible by more than one thread. We present a variant of the Cartesian abstraction that breaks the dependencies between the threads but doesn't break the dependencies between the shared state space and the private state space of a thread. To introduce that abstraction, we need a concrete domain, an abstract domain and a Galois connection between them.

$D := \mathfrak{P}(\text{States})$ is the set underlying the concrete lattice,
$D^\# := (\mathfrak{P}(\text{Glob} \times \text{Loc}))^n$ is the set underlying the abstract lattice,
$\alpha_{\text{mc}} :\ D \to D^\#,$
$\qquad S \mapsto (\pi_{0,i}(S))_{i=1}^n$
is the abstraction map, which projects a set of program states to the tuple of

sets of thread states, so that the i^{th} component of a tuple contains exactly the states of the i^{th} thread that occur in the set of program states. The subscript mc abbreviates multithreaded Cartesian.

$\gamma_{\text{mc}}: \quad D^\# \to D,$
$\qquad T \mapsto \{(g,l) \mid \forall\, i \in \mathbb{N}_n : (g,l_i) \in T_i\}$

is the concretization map that combines a tuple of sets of thread states to a set of program states by putting only those thread states together that have equal global part.

The ordering on the concrete domain D is inclusion, the least upper bound is the union $\cup$, the greatest lower bound is the intersection $\cap$, and X^c is the complement of X in States.

The ordering on the abstract domain $D^\#$ is the product ordering, i.e. $T \sqsubseteq T'$ if and only if $T_i \subseteq T'_i$ for all $i \in \mathbb{N}_n$. The least upper bound $\sqcup$ is componentwise union, the greatest lower bound $\sqcap$ is componentwise intersection. Thus the abstract lattice is complete. The bottom element is the tuple of empty sets $\bot = (\emptyset)_{i=1}^n$.

Proposition and Definition 1.5.1. The pair of maps $(\alpha_{\text{mc}}, \gamma_{\text{mc}})$ is a Galois connection. This pair is called the *multithreaded Cartesian Galois connection*, α_{mc} the *multithreaded Cartesian abstraction function*, γ_{mc} the *multithreaded Cartesian concretization function*.

Proof. Let's show that for all $S \in D, T \in D^\#$ we have

$$\alpha_{\text{mc}}(S) \sqsubseteq T \text{ iff } S \subseteq \gamma_{\text{mc}}(T)\,.$$

"$\Rightarrow$": Let $(g,l) \in S$. Let $T' = \alpha_{\text{mc}}(S)$. Then by definition of α_{mc} we have for all $i \in \mathbb{N}_n$ that $(g,l_i) \in T'_i \subseteq T_i$. So $(g,l) \in \gamma_{\text{mc}}(T)$ by definition of γ_{mc}.
"$\Leftarrow$": Let $T' = \alpha_{\text{mc}}(S)$. Fix some $i \in \mathbb{N}_n$ and let $(g,l_i) \in T'_i$. By definition of α_{mc} for each other $j \neq i$ there is an l_j so that the tuple containing all these elements $(g,(l_j)_{j=1}^n)$ is in S. But $S \subseteq \gamma_{\text{mc}}(T)$, so $(g,(l_j)_{j=1}^n) \in \gamma_{\text{mc}}(T)$. By definition of γ_{mc} we have $(g,l_i) \in T_i$. So $T'_i \subseteq T_i$. Since i was arbitrarily chosen, we have $T' \sqsubseteq T$ by definition of $\sqsubseteq$. □

The main task of an implementation of abstract fixpoint checking with multithreaded Cartesian Galois connection is computation of the least fixpoint of $\lambda y.\, \alpha_{\text{mc}}(\text{init} \cup \text{post} \circ \gamma_{\text{mc}}(y)) \in D^\# \to D^\#$. This function is monotone. Computing the least fixpoint of a monotone function f in a complete lattice $(L, \preceq)$ with bottom $\bot$ usually requires computing some (or all or even intermediate) elements of the chain $\bot \preceq f(\bot) \preceq f(f(\bot)) \preceq f(f(f(\bot))) \preceq \dots$. An approximate but nevertheless useful (at least in the finite-state case) upper bound on the number of iterations an implementation has to make is the length of such a chain.

Definition 1.5.2. The *height* of a poset $(X, \preceq)$ is the supremum of cardinalities of its chains:

$$\mathfrak{h}(X, \preceq) := \sup\{|C| \mid C \text{ is a chain in } (X, \preceq)\}\,.$$

Proposition 1.5.3. The height of the power order is linear in the number of components and is linear in the height of every component. More precisely:

Let $(X, \preceq)$ be a nonempty (i.e. $X \neq \emptyset$) poset of finite height, let $n \in \mathbb{N}_0$ and let $\preceq_n = \{(x, x') \in (X^n)^2 \mid \forall\, i \in \mathbb{N}_n : x_i \preceq x'_i\}$ be the n^{th} power order of $\preceq$. Then

$$\mathfrak{h}(X^n, \preceq_n) \;=\; (\mathfrak{h}(X, \preceq) - 1)n + 1\,.$$

Idea. For "$\leq$", map a chain in the product to chains in the components and count their total lengths. For "$\geq$", construct a chain with thus many elements.
Proof. For $n = 0$ the 0^{th} power of a set contains exactly one element. For $n = 1$ there is nothing to show. From now on let $n \geq 2$. Let X be the domain of the underlying order $\preceq$. We prove "$\leq$" and "$\geq$" separately.

"$\leq$": For a chain K of $(X^n, \preceq_n)$, we shall show that $|K| \leq (\mathfrak{h}(X, \preceq)-1)n+1$. Since X is nonempty, $\mathfrak{h}(X, \preceq) \geq 1$ and so the right-hand-side of the inequality is at least 1. Thus if K is empty or a singleton, it satisfies the inequality. From now on let K have at least two elements. Notice that for each $i \in \mathbb{N}_n$, the set $\pi_i(K)$ is a chain in $(X, \preceq)$ and hence finite. From $K \subseteq \prod_{i=1}^n \pi_i(K)$ follows that K is finite. Enumerate elements of K in ascending order by natural numbers from 1 to m:

$$x^{(1)} \prec_n \ldots \prec_n x^{(m)}\,.$$

For each position $1 \leq j < m$ there is a least one changing component $c(j) \in \mathbb{N}_n$, i.e. with $x^{(j)}_{c(j)} \prec x^{(j+1)}_{c(j)}$. Consider the map

$$f : (K \setminus (\max K)) \to \dot{\bigcup_{i=1}^n} \{i\} \times (\pi_i(K) \setminus (\max \pi_i(K)))\,, \quad x^{(j)} \mapsto (c(j), x^{(j)}_{c(j)})\,.$$

This map is well-defined. We claim that f is one-to-one. Since otherwise there would exist positions $j, k \in \mathbb{N}^+$ such that $j < k \leq m$ and $f(x^{(j)}) = (c(j), x^{(j)}_{c(j)}) = (c(k), x^{(k)}_{c(k)}) = f(x^{(k)})$; then $c(j) = c(k)$ and thus $x^{(k)}_{c(k)} = x^{(j)}_{c(j)} \prec x^{(j+1)}_{c(j)} \preceq \ldots$ [from $j + 1 \leq k$ and the definition of the product order] $\preceq x^{(k)}_{c(j)} = x^{(k)}_{c(k)}$, giving a contradiction. Since f is one-to-one, its domain is not larger than its image, i.e. $|K \setminus (\max K)| \leq \sum_{i=1}^n |\pi_i(K) \setminus (\max \pi_i(K))| \leq (\mathfrak{h}(X, \preceq) - 1)n$. Thus $|K| \leq (\mathfrak{h}(X, \preceq) - 1) + 1$.

"$\geq$": If $\mathfrak{h}(X, \preceq) = 1$, then $|X| \geq 1$, thus $|X^n| \geq 1$, and so $\mathfrak{h}(X^n, \preceq_n) \geq 1 = (\mathfrak{h}(X, \preceq)-1)n+1$. From now on let $\mathfrak{h}(X, \preceq) \geq 2$. Let K be any chain of $(X, \preceq)$ of maximal cardinality. Enumerate the elements of K by natural numbers from 1 to $m = \mathfrak{h}(X, \preceq)$ in ascending order:

$$x^{(1)} \prec \ldots \prec x^{(m)}\,.$$

Let

$$\begin{aligned} \bar{K} := \{y \in K^n \mid \exists\, i \in \mathbb{N}_n : y_i \prec x^{(m)} \text{ and} \\ (\forall\, j \in \mathbb{N}_n \text{ with } j < i : y_j = x^{(m)}) \text{ and} \\ (\forall\, j \in \mathbb{N}_n \text{ with } j > i : y_j = x^{(1)})\}\,. \end{aligned}$$

First we claim that $\bar{K}$ is a chain in $(X^n, \preceq_n)$.

To prove that, let $y, y' \in \bar{K}$. For y (resp. y') there is an index i (resp. i') such that y_i (resp. $y'_{i'}$) is not $x^{(m)}$, the components at positions less than i (resp. i') are all $x^{(m)}$ and components at positions greater that i (resp. i') are all $x^{(1)}$. We distinguish three cases.

1. $i < i'$. Then for $j \in \mathbb{N}^+$, $j < i'$ we have $y_j \preceq x^{(m)} = y'_j$. And for $j \in \mathbb{N}^+$, $i' \leq j \leq n$ we have $y_j = x^{(1)} \leq y'_j$. Thus $y \preceq_n y'$.
2. $i' < i$. Analogously. For $j \in \mathbb{N}^+$, $j < i$ we have $y'_j \preceq x^{(m)} = y_j$. And for $j \in \mathbb{N}^+$, $i \leq j \leq n$ we have $y'_j = x^{(1)} \leq y_j$. Thus $y' \preceq_n y$.
3. $i = i'$. The i^{th} components of y and y' are comparable, all the other components are equal. So y and y' are comparable.

Thus $\bar{K}$ is a chain.

Now we claim that $|\bar{K}| \geq (m-1)n$.

To prove that, let $f : \mathbb{N}_n \times \mathbb{N}_{m-1} \to \bar{K}$, where $f(i,k)$ is the tuple $y \in \bar{K}$ in which for all $j \in \mathbb{N}^+$ with $j < i$ we have $y_j = x^{(m)}$, for all $j \in \mathbb{N}^+$ with $i < j \leq n$ we have $y_j = x^{(1)}$ and $y_i = x^{(k)}$. To prove that f is one-to-one, take different $(i,k), (i',k') \in \mathbb{N}_n \times \mathbb{N}_{m-1}$. There are three cases.

1. $i = i'$ and $k \neq k'$. Then by construction $(f(i,k))_i = x^{(k)} \neq x^{(k')} = (f(i',k'))_{i'} = (f(i',k'))_i$.
2. $i < i'$. Then $(f(i,k))_i \prec x^{(m)} = (f(i',k'))_i$.
3. $i' < i$. Then $(f(i',k'))_{i'} \prec x^{(m)} = (f(i,k))_{i'}$.

In any of these cases we conclude $f(i,k) \neq f(i',k')$. So f is one-to-one and thus its image is not smaller than its domain: $|\bar{K}| \geq n(m-1)$.

Notice that $(x^{(m)})_{i=1}^n$ is not an element of $\bar{K}$ but is greater than any element of $\bar{K}$. Thus $\bar{K} \dot\cup \{(x^{(m)})_{i=1}^n\}$ is a chain in $(X^n, \preceq_n)$ with at least $n(m-1)+1 = (\mathfrak{h}(X, \preceq) - 1)n + 1$ elements. □

Now we can compare the abstract and the concrete lattices by cardinality and height, showing that the abstract is (also asymptotically) smaller.

Proposition 1.5.4. Let $g = |\text{Glob}| \in \mathbb{N}^+$ be constant and $l = |\text{Loc}| \in \mathbb{N}^+$. Then:

1. $|D| = 2^{gl^n}$;
2. $|D^\#| = 2^{gln}$;
3. $\mathfrak{h}(D, \subseteq) = gl^n + 1$;
4. $\mathfrak{h}(D^\#, \sqsubseteq) = gln + 1$;
5. If $l \geq 2$ then $|D^\#| \leq |D|$;
6. For a constant $l \geq 2$ and $n \to \infty$ or for a constant $n \geq 2$ and $l \to \infty$ we have $\lim \dfrac{|D^\#|}{|D|} = 0$;
7. If $l \geq 2$ then $\mathfrak{h}(D^\#, \sqsubseteq) \leq \mathfrak{h}(D, \subseteq)$;
8. For a constant $l \geq 2$ and $n \to \infty$ or for a constant $n \geq 2$ and $l \to \infty$ we have $\lim \dfrac{\mathfrak{h}(D^\#, \sqsubseteq)}{\mathfrak{h}(D, \subseteq)} = 0$.

Proof.

1. $|D| = |\mathfrak{P}(\text{Glob} \times \text{Loc}^n)| = 2^{|\text{Glob} \times \text{Loc}^n|} = 2^{gl^n}$.
2. $|D^\#| = |(\mathfrak{P}(\text{Glob} \times \text{Loc}))^n| = |\mathfrak{P}(\text{Glob} \times \text{Loc})|^n = (2^{|\text{Glob} \times \text{Loc}|})^n = 2^{gln}$.
3. $\mathfrak{h}(D, \subseteq) = \mathfrak{h}(\mathfrak{P}(\text{Glob} \times \text{Loc}^n), \subseteq) = |\text{Glob} \times \text{Loc}^n| + 1 = gl^n + 1$.
4. From Prop. 1.5.3 follows that $\mathfrak{h}(D^\#, \sqsubseteq) = \mathfrak{h}((\mathfrak{P}(\text{Glob} \times \text{Loc}))^n, \sqsubseteq) = (\mathfrak{h}(\mathfrak{P}(\text{Glob} \times \text{Loc}), \subseteq) - 1)n + 1 = (gl + 1 - 1)n + 1 = gln + 1$.
5. For $l \geq 2$ and $n \geq 1$ we always have $ln \leq l^n$.
6. $\dfrac{|D^\#|}{|D|} = 2^{g(ln - l^n)} \to 0$ if $l \geq 2$ and $n \to \infty$ or if $n \geq 2$ and $l \to \infty$.
7. For $g \geq 1$, $l \geq 2$ and $n \geq 1$ we always have $ln \leq l^n$, thus $gln + 1 \leq gl^n + 1$.

8. For $l \geq 2$ and $n \to \infty$ or for $n \geq 2$ and $l \to \infty$ we have $\lim \dfrac{gln+1}{gl^n+1} = \lim \dfrac{gln}{gl^n+1} + \lim \dfrac{1}{gl^n+1} = \lim \dfrac{1}{\frac{l^n}{ln} + \frac{1}{gln}} + 0 = 0.$

□

Since $\mathfrak{h}(D^\#, \sqsubseteq)$ is asymptotically strictly smaller that $\mathfrak{h}(D, \subseteq)$, a naive least fixpoint computation in $D^\#$ is in general much faster than a naive least fixpoint computation in D. Given that D, $D^\#$, init and post are dependent on the underlying program (which is given by the user), we expect that computing the least fixpoint of $\lambda x.\, \text{init} \cup \text{post}(x)$ in D is in the worst case always slower than naively computing the least abstract fixpoint of $\lambda y.\, \alpha_{\text{mc}}(\text{init} \cup \text{post} \circ \gamma_{\text{mc}}(y))$ in $D^\#$.

1.6 Multithreaded Cartesian abstraction reviewed

For the concrete and abstract lattices there are isomorphic lattices such that the induced multithreaded Cartesian Galois connection between the new pair of lattices can be expressed in terms of the well-known standard Cartesian Galois connection. This simple view enables carrying over the known techniques for the standard Cartesian Galois connection to the multithreaded one.

For that, consider $\overline{D} = (\text{Glob} \to \mathfrak{P}(\text{Loc}^n))$ and $\overline{D^\#} = (\text{Glob} \to (\mathfrak{P}(\text{Loc}))^n)$. Both lattices $\overline{D}$ and $\overline{D^\#}$ are equipped with the pointwise order, i.e. for $f_1, f_2 \in \overline{D}$ (resp. $\in \overline{D^\#}$), we have that f_1 is less than or equal to f_2 if for each $g \in \text{Glob}$, $f_1(g) \subseteq f_2(g)$ (resp. $f_1(g)$ is less than or equal to $f_2(g)$ in the product order). Consider the maps

$$\begin{array}{rll}
\phi\colon & D \to \overline{D}\,, & S \mapsto \{(g, \{l \mid (g,l) \in S\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\}\,, \\
\psi\colon & D^\# \to \overline{D^\#}\,, & (T_i)_{i=1}^n \mapsto \{(g, (\{l \mid (g,l) \in T_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\}\,, \\
\overline{\alpha_{\text{mc}}}\colon & \overline{D} \to \overline{D^\#}\,, & f \mapsto \{(g, \alpha_{\text{c}}(f(g))) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\}\,, \\
\overline{\gamma_{\text{mc}}}\colon & \overline{D^\#} \to \overline{D}\,, & f \mapsto \{(g, \gamma_{\text{c}}(f(g))) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\}\,,
\end{array}$$

where $(\alpha_{\text{c}}, \gamma_{\text{c}})$ is the standard Cartesian Galois connection as in (1.2) between the domains $\mathfrak{P}(\text{Loc}^n)$ and $(\mathfrak{P}(\text{Loc}))^n$.

A *complete isomorphism* is a bijection which is a complete join-morphism and a complete meet-morphism.

Theorem 1.6.1. ϕ is a complete isomorphism between D and $\overline{D}$.

Idea. Intuition from the type theory: $(\text{Glob} \times X) \to \text{Bool}$ is an uncurrying of $\text{Glob} \to (X \to \text{Bool})$. No approximation happens on the shared part by union and intersection.

Proof. We show the following.

- ϕ is one-to-one. Let $\phi(S) = \phi(T)$ for some $S, T \in D$. Then for all $g \in \text{Glob}$ we have $\{l \mid (g,l) \in S\} = \{l \mid (g,l) \in T\}$. Thus for all $g \in \text{Glob}$ and all l we have $(g,l) \in S \Leftrightarrow (g,l) \in T$. So $S = T$.
- ϕ is onto. Let $f \in \overline{D}$. Define $S := \{(g,l) \mid g \in \text{Glob} \text{ and } l \in f(g)\}$. Then $\phi(S) = \{(g, \{l \mid (g,l) \in S\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = \{(g, \{l \mid l \in f(g)\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = \{(g, f(g)) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = f$.

- ϕ is a complete join-morphism. Let $S \subseteq D$. Then $\phi(\bigcup S) = \{(g, \{l \mid (g,l) \in \bigcup S\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = \{(g, \{l \mid (g,l) \in T \text{ for some } T \in S\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = \{(g, \bigcup_{T \in S} \{l \mid (g,l) \in T\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\}$ = (pointwise union of maps $\{(g, \{l \mid (g,l) \in T\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\}$ over $T \in S$) = (pointwise union of maps $\phi(T)$ where $T \in S$).
- ϕ is a complete meet-morphism. Let $S \subseteq D$. Then $\phi(\bigcap S) = \{(g, \{l \mid (g,l) \in \bigcap S\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = \{(g, \{l \mid (g,l) \in T \text{ for each } T \in S\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\} = \{(g, \bigcap_{T \in S} \{l \mid (g,l) \in T\}) \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\}$ = (pointwise intersection of maps $\{(g, \{l \mid (g,l) \in T\} \in \text{Glob} \times \mathfrak{P}(\text{Loc}^n)\}$ over $T \in S$) = pointwise intersection of maps $\phi(T)$ with $T \in S$.

□

Theorem 1.6.2. ψ is a complete isomorphism between $D^\#$ and $\overline{\overline{D^\#}}$

Idea. Intuition from the type theory: $(\text{Glob} \times X) \to \text{Bool}$ is an uncurrying of $\text{Glob} \to (X \to \text{Bool})$. No approximation happens on the shared part by join and meet.

Proof. We show:

- ψ is one-to-one. Let $S = (S_i)_{i=1}^n, T = (T_i)_{i=1}^n \in D^\#$ with $\psi(S) = \psi(T)$. Then for all $g \in \text{Glob}$ and all $i \in \mathbb{N}_n$ we have $\{l \mid (g,l) \in S_i\} = \{l \mid (g,l) \in T_i\}$. Thus for all $i \in \mathbb{N}_n$ and $g \in \text{Glob}$ and $l \in \text{Loc}$ we have $(g,l) \in S_i \Leftrightarrow (g,l) \in T_i$. Thus for all $i \in \mathbb{N}_n$ we have $S_i = T_i$.
- ψ is onto. Let $f \in \overline{D^\#}$. Define $S_i := \{(g,l) \in \text{Glob} \times \text{Loc} \mid l \in (f(g))_i\}$ and $S := (S_i)_{i=1}^n$. Then for each $g \in \text{Glob}$ we have $\psi(S)(g) = (\{l \mid (g,l) \in S_i\})_{i=1}^n = ((f(g))_i)_{i=1}^n = f(g)$. Thus $\psi(S) = f$.
- ψ is a complete join-morphism. Let $S \subseteq D^\#$. Then $\psi(\bigsqcup S) = \{(g, (\{l \mid (g,l) \in (\bigsqcup S)_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\} = \{(g, (\{l \mid (g,l) \in T_i \text{ for some } T \in S\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\} = \{(g, (\bigcup_{T \in S} \{l \mid (g,l) \in T_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\}$ = (pointwise join of maps $\{(g, (\{l \mid (g,l) \in T_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\}$ over $T \in S$) = (pointwise join of maps $\psi(T)$ over $T \in S$).
- ψ is a complete meet-morphism. Let $S \subseteq D^\#$. Then $\psi(\bigsqcap S) = \{(g, (\{l \mid (g,l) \in (\bigsqcap T)_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\} = \{(g, (\{l \mid (g,l) \in T_i \text{ for all } T \in S)_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\} = \{(g, (\bigcap_{T \in S} \{l \mid (g,l) \in T_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\}$ = (pointwise meet of maps $\{(g, (\{l \mid (g,l) \in T_i\})_{i=1}^n) \in \text{Glob} \times (\mathfrak{P}(\text{Loc}))^n\}$ over $T \in S$) = (pointwise meet of maps $\psi(T)$ over $T \in S$).

□

Multithreaded Cartesian Galois connection is isomorphic to the new one:

Theorem 1.6.3. $\overline{\alpha_{\text{mc}}} = \psi \circ \alpha_{\text{mc}} \circ \phi^{-1}$ and $\overline{\gamma_{\text{mc}}} = \phi \circ \gamma_{\text{mc}} \circ \psi^{-1}$.

Proof. Let $S \in D$. Then $\overline{\alpha_{\text{mc}}}(\phi(S)) = \lambda g.\, (\{l_i \mid (g,l) \in S\})_{i=1}^n = \psi(\alpha_{\text{mc}}(S))$. So $\overline{\alpha_{\text{mc}}} \circ \phi = \psi \circ \alpha_{\text{mc}}$. Multiply with ϕ^{-1} from the right.
Now let $T \in D^\#$. Then $\overline{\gamma_{\text{mc}}}(\psi(T)) = \lambda g.\, \{l \mid \forall\, i \in \mathbb{N}_n : (g, l_i) \in T_i\} = \phi(\gamma_{\text{mc}}(T))$. So $\overline{\gamma_{\text{mc}}} \circ \psi = \phi \circ \gamma_{\text{mc}}$. Multiply with ψ^{-1} from the right. □

1.7 Thread-modular = multithreaded Cartesian

Now we prove precision equivalence between the Flanagan-Qadeer algorithm and the abstract fixpoint checking with multithreaded Cartesian abstraction.

For our tiny example from section 1.4, the abstract fixpoint with multithreaded Cartesian abstraction, i.e. $\mathrm{lfp}(\lambda y.\, \alpha_{\mathrm{mc}}(\mathrm{init} \cup \mathrm{post} \circ \gamma_{\mathrm{mc}}(y)))$, is

$$(\{(0,A),(0,B),(1,A),(1,B)\},\ \ \{(0,C),(0,D),(1,D)\})\ ,$$

which coincides with $(\mathcal{R}_1, \mathcal{R}_2)$ computed by the Flanagan-Qadeer algorithm. It turns out that this coincidence is not by pure accident.

Fix a multithreaded program and consider the set tuple $((\mathcal{R}_i, \mathcal{G}_i))_{i\in\mathbb{N}_n}$ that represents the least fixpoint of the Flanagan-Qadeer inference rules.

We are going to analyze a simpler system FQ' of inference rules, which doesn't speak about the sets $\mathcal{G}$:

$$\text{INIT}\ \frac{}{\mathrm{init}_i \subseteq \mathcal{R}_i} \qquad \text{STEP'}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,l) \to_i (g',l')}{(g',l') \in \mathcal{R}_i}$$

$$\text{ENV'}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,l2) \in \mathcal{R}_j \qquad (g,l2) \to_j (g',l2')}{(g',l) \in \mathcal{R}_i}\ i \neq j$$

As before, $\mathrm{init}_i = \{(g, l_i) \mid (g,l) \in \mathrm{init}\}$.

To justify our analysis of FQ' instead of FQ, we prove the

Proposition 1.7.1. The least fixpoints of FQ and FQ' define the same sets of thread states. Formally:
If $(\mathcal{R}_i, \mathcal{G}_i)_{i=1}^n$ is the least fixpoint of FQ and $(\mathcal{R}'_i)_{i=1}^n$ is the least fixpoint of FQ', then $(\mathcal{R}_i)_{i=1}^n = (\mathcal{R}'_i)_{i=1}^n$.

Idea. Syntactic transformation between the proof trees: ENV' $\approx$ ENV+STEP.
Proof. By Thm. 1.2.3 each element of $\mathcal{R}_i$, $\mathcal{R}'_i$, $\mathcal{G}_i$ ($i \in \mathbb{N}_n$) has a finite derivation tree. First we show that $(\mathcal{R}_i)_{i=1}^n \sqsubseteq (\mathcal{R}'_i)_{i=1}^n$. For that, take any element of $\mathcal{R}_i$ for some $i \in \mathbb{N}_n$, take its derivation tree in FQ, find the applications of the ENV rule:

$$\text{ENV}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,g') \in \mathcal{G}_j}{(g',l) \in \mathcal{R}_i}\ i \neq j.$$

The statement "$(g,g') \in \mathcal{G}_j$" has a derivation tree whose root is inevitably generated by STEP. Together those rules look then as follows:

$$\text{ENV}\ \frac{(g,l) \in \mathcal{R}_i \qquad \text{STEP}\ \dfrac{(g,\bar{l}) \in \mathcal{R}_j \qquad (g,\bar{l}) \to_j (g',\bar{l}')}{(g,g') \in \mathcal{G}_j}}{(g',l) \in \mathcal{R}_i}\ i \neq j.$$

Combine these two rules into one rule ENV':

$$\text{ENV'}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,\bar{l}) \in \mathcal{R}_j \qquad (g,\bar{l}) \to_j (g',\bar{l}')}{(g',l) \in \mathcal{R}_i}\ i \neq j.$$

By repeating the described process for all ENV rules the propositions "$(g,g') \in \mathcal{G}_i$" get unused. Thus replacing STEP by STEP' gives a valid inference tree in FQ' with the same root.

To prove $(\mathcal{R}'_i)_{i=1}^n \sqsubseteq (\mathcal{R}_i)_{i=1}^n$, take any element of $\mathcal{R}'_i$ for some $i \in \mathbb{N}_n$. Take its derivation tree and replace each application of the ENV' rule

$$\text{ENV'}\ \frac{(g,l) \in \mathcal{R}_i \qquad (g,\bar{l}) \in \mathcal{R}_j \qquad (g,\bar{l}) \to_j (g',\bar{l}')}{(g',l) \in \mathcal{R}_i}\ i \neq j$$

by the following two rules:

$$\text{ENV}\ \frac{(g,l)\in\mathcal{R}_i \qquad \overset{\text{STEP}}{\dfrac{(g,\bar{l})\in\mathcal{R}_j \qquad (g,\bar{l})\to_j(g',\bar{l}')}{(g,g')\in\mathcal{G}_j}}}{(g',l)\in\mathcal{R}_i}\ i\neq j.$$

Then rename the STEP' rules into STEP to obtain a valid derivation tree in FQ. The new tree has the same root as the old one. □

So it doesn't matter whether we speak about the sets of thread states coming from the least fixpoint of FQ or of FQ'.

Theorem 1.7.2 (Thread-modular = multithreaded Cartesian)**.** Thread-modular model-checking has the same precision as abstract fixpoint checking with the multithreaded Cartesian Galois connection. Formally:

$$(\mathcal{R}_i)_{i=1}^n \;=\; \mathrm{lfp}(\lambda y.\,\alpha_{\mathrm{mc}}(\mathrm{init}\cup\mathrm{post}\circ\gamma_{\mathrm{mc}}(y)))\,.$$

Idea. We have to show that two maps have the same least fixpoints. It suffices to show that the least fixpoint of one map is a postfixpoint of the other map and vice versa.

Proof. The abstraction and concretization maps α_{mc} and γ_{mc} form a Galois connection by Prop. 1.5.1. Thus α_{mc} is a join-morphism by Prop. 3, part 2 of [1]. So rewrite the function as $F = \lambda y.\,\alpha_{\mathrm{mc}}(\mathrm{init})\sqcup\alpha_{\mathrm{mc}}\circ\mathrm{post}\circ\gamma_{\mathrm{mc}}(y) = \lambda y.\,(\mathrm{init}_i)_{i=1}^n\sqcup\alpha_{\mathrm{mc}}\circ\mathrm{post}\circ\gamma_{\mathrm{mc}}(y)$.

Moreover, Prop. 2 of [1] shows that α_{mc} and γ_{mc} are monotone. Since post is also monotone, F is monotone. By the fixpoint theorem of Tarski (Thm. 1 in [16]) F has a least fixpoint, call it $X=(X_i)_{i=1}^n$. Let $Y=(Y_i)_{i=1}^n$ be the least fixpoint of FQ'.

"$X\sqsubseteq Y$": Since X is the least of all postfixpoints of F, it suffices to show that Y is a postfixpoint of F, i.e. that $F(Y)\sqsubseteq Y$. So let $i\in\mathbb{N}_n$ and $(g,l)\in(F(Y))_i$. There are two cases.

- Case $(g,l)\in\mathrm{init}_i$. Then $(g,l)\in(\mathbb{D}_{\mathrm{FQ}'}(\bot))_i$, it can be obtained by a single application of rule INIT. Thm. 1.2.3 implies $(g,l)\in Y_i$.
- Case $(g,l)\in(\alpha_{\mathrm{mc}}\circ\mathrm{post}\circ\gamma_{\mathrm{mc}}(Y))_i$. Then there is some $\mathbf{l}\in\mathrm{Loc}^n$ such that $(g,\mathbf{l})\in\mathrm{post}\circ\gamma_{\mathrm{mc}}(Y)$ and $l=\mathbf{l}_i$. Thus there is some thread $j\in\mathbb{N}_n$ and a thread state $(\bar{g},\bar{l})$ such that $(\bar{g},\bar{l})\to_j(g,\mathbf{l}_j)$ and $(\bar{g},\mathbf{l}[j\mapsto\bar{l}])\in\gamma_{\mathrm{mc}}(Y)$. Thus for all $k\in\mathbb{N}_n\setminus\{j\}$ we have $(\bar{g},\mathbf{l}_k)\in Y_k$ and $(\bar{g},\bar{l})\in Y_j$, both being generated by finite derivation trees. There are two cases.
  - Case $i=j$. The STEP' rule

$$\text{STEP'}\ \frac{(\bar{g},\bar{l})\in\mathcal{R}_i \qquad (\bar{g},\bar{l})\to_i(g,l)}{(g,l)\in\mathcal{R}_i}$$

    ensures that $(g,l)\in Y_i$.
  - Case $i\neq j$. The ENV' rule

$$\text{ENV'}\ \frac{(\bar{g},l)\in\mathcal{R}_i \qquad (\bar{g},\bar{l})\in\mathcal{R}_j \qquad (\bar{g},\bar{l})\to_j(g,\mathbf{l}_j)}{(g,l)\in\mathcal{R}_i}$$

    ensures that $(g,l)\in Y_i$.

"$Y \sqsubseteq X$": Since Y is the least of all postfixpoints of the derivation operator of FQ', it suffices to show that X is a postfixpoint of the derivation operator of FQ', i.e. that $\mathbb{D}_{\mathrm{FQ}'}(X) \sqsubseteq X$. So let $i \in \mathbb{N}_n$ and $(g,l) \in (\mathbb{D}_{\mathrm{FQ}'}(X))_i$. Then there is a rule $(A, i, (g,l))$ of FQ' such that $A \sqsubseteq X$. Look at the rule type.

- The rule is INIT. Then $(g,l) \in \mathrm{init}_i$. Since X is a fixpoint, it is especially a postfixpoint, i.e. $F(X) \sqsubseteq X$. Thus $\alpha_{\mathrm{mc}}(\mathrm{init}) \sqsubseteq X$ and so $\mathrm{init}_i \subseteq X_i$, thus $(g,l) \in X_i$.
- The rule is STEP'. Then there is some $(\bar{g},\bar{l}) \in X_i$ such that $(\bar{g},\bar{l}) \to_i (g,l)$. Notice that $X = F(X)$ and thus $X_k = \{(\tilde{g},\tilde{l}_k) \mid (\tilde{g},\tilde{l}) \in \mathrm{init} \cup \mathrm{post} \circ \gamma_{\mathrm{mc}}(X)\}$ for any $k \in \mathbb{N}_n$. From $(\bar{g},\bar{l}) \in X_i$ we obtain that there is some $\mathbf{l} \in \mathrm{Loc}^n$ such that for all $k \in \mathbb{N}_n$ we have $(\bar{g},\mathbf{l}_k) \in X_k$ and $\mathbf{l}_i = \bar{l}$. Then $(\bar{g},\mathbf{l}) \in \gamma_{\mathrm{mc}}(X)$. Then $(g,\mathbf{l}[i \mapsto l]) \in \mathrm{post} \circ \gamma_{\mathrm{mc}}(X)$. Then $(g,l) \in (\alpha_{\mathrm{mc}}(\mathrm{post} \circ \gamma_{\mathrm{mc}}(X)))_i \subseteq (\alpha_{\mathrm{mc}}(\mathrm{init} \cup \mathrm{post} \circ \gamma_{\mathrm{mc}}(X)))_i = (F(X))_i = X_i$.
- The rule is ENV'. Then there is some $j \in \mathbb{N}_n \setminus \{i\}$, $\bar{g} \in \mathrm{Glob}$, $\tilde{l},\tilde{l}' \in \mathrm{Loc}$ such that $(\bar{g},l) \in X_i$, $(\bar{g},\tilde{l}) \in X_j$ and $(\bar{g},\tilde{l}) \to_j (g,\tilde{l}')$. Notice that $X = F(X)$ and thus $X_k = \{(\hat{g},\hat{\mathbf{l}}_k) \mid (\hat{g},\hat{\mathbf{l}}) \in \mathrm{init} \cup \mathrm{post} \circ \gamma_{\mathrm{mc}}(X)\}$ for any $k \in \mathbb{N}_n$. So there are tuples $\bar{\mathbf{l}},\tilde{\mathbf{l}} \in \mathrm{Loc}^n$ such that for all $k \in \mathbb{N}_n$ we have $(\bar{g},\bar{\mathbf{l}}_k),(\bar{g},\tilde{\mathbf{l}}_k) \in X_k$ and $\bar{\mathbf{l}}_i = l$ and $\tilde{\mathbf{l}}_j = \tilde{l}$. Then $(\bar{g},\bar{\mathbf{l}}[j \mapsto \tilde{l}]) \in \gamma_{\mathrm{mc}}(X)$. Then $(g,\bar{\mathbf{l}}[j \mapsto \tilde{l}']) \in \mathrm{post} \circ \gamma_{\mathrm{mc}}(X)$. Thus $(g,l) \in (\alpha_{\mathrm{mc}}(\mathrm{post} \circ \gamma_{\mathrm{mc}}(X)))_i \subseteq (\alpha_{\mathrm{mc}}(\mathrm{init} \cup \mathrm{post} \circ \gamma_{\mathrm{mc}}(X)))_i = (F(X))_i = X_i$.

□

1.8 Abstraction and approximation*

To speak further about multithreaded Cartesian abstraction, we first show that for fixpoint checking it doesn't matter whether Galois connections or upper closure operators are considered. A reader who feels comfortable with upper closure operators and Galois connections may skip this part during first reading.

When speaking about a particular program analysis technique which doesn't operate with concrete states but with some abstract objects that live in another domain, it is natural to ask whether for the computation on the abstract domain there is a corresponding computation on the concrete domain which produces an equivalent result. Although it is impractical to operate with concrete sets of states, it is nevertheless helpful to visualize the abstract computation by comparing it to a concrete one.

So assume a Galois connection (α,γ) between some complete lattices $(D,\preceq)$ (concrete domain, usually the power set of the set of states) and $(D^\#,\sqsubseteq)$ (abstract domain) and some monotone operator $f : D \to D$ on the concrete domain. For instance, f could be defined by $f(x) = \mathrm{init} \cup \mathrm{post}(x)$ for a program with successor map post and initial states init. A natural choice for the approximation step on the concrete domain would be $\rho = \gamma \circ \alpha$ because of

Proposition 1.8.1. ρ is an upper closure operator.

Proof. Prop. 2 in [1] says that α and γ are monotone and that $\gamma \circ \alpha$ is extensive and that $\alpha \circ \gamma$ is reductive. So ρ is monotone and extensive. To show idempotence, let $x \in D$. Extensivity implies $\rho(\rho(x)) \succeq \rho(x)$. Now notice that $\rho \circ \rho(x) = (\gamma \circ \alpha) \circ (\gamma \circ \alpha)(x) = \gamma(\underbrace{\alpha \circ \gamma}_{\text{reductive}}(\alpha(x))) \preceq \gamma(\alpha(x)) = \rho(x)$. □

The relationship between the least fixpoint of $\rho \circ f$ in D and the least fixpoint of $\alpha \circ f \circ \gamma$ in $D^\#$ is clarified by the following

Theorem 1.8.2. The least fixpoint of a function under a Galois connection in the abstract lattice coincides up to abstraction/concretization with the least fixpoint of the function under the corresponding upper closure operator in the concrete lattice. Formally:

$$\begin{array}{rcl} \mathrm{lfp}(\gamma \circ \alpha \circ f) & = & \gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma) \quad , \\ \alpha \circ \mathrm{lfp}(\gamma \circ \alpha \circ f) & = & \mathrm{lfp}(\alpha \circ f \circ \gamma) \quad . \end{array}$$

Idea. Empirically one discovers equality for finite iterates in the naive computations of the least fixpoints (assuming for simplicity that concretization is zero-strict). The real proof is a consequence of Tarski's fixpoint theorem and properties of Galois connections.

Proof. It suffices to show

1. $\mathrm{lfp}(\gamma \circ \alpha \circ f) \preceq \gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma)$, which is equivalent to $\alpha \circ \mathrm{lfp}(\gamma \circ \alpha \circ f) \sqsubseteq \mathrm{lfp}(\alpha \circ f \circ \gamma)$,
2. $\mathrm{lfp}(\gamma \circ \alpha \circ f) \succeq \gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma)$,
3. $\alpha \circ \mathrm{lfp}(\gamma \circ \alpha \circ f) \sqsupseteq \mathrm{lfp}(\alpha \circ f \circ \gamma)$.

Since the abstraction and concretization maps are monotone, so are $\gamma \circ \alpha \circ f$ and $\alpha \circ f \circ \gamma$. According to Tarski's fixpoint theorem,

$$\mathrm{lfp}(\alpha \circ f \circ \gamma) = \inf\{y \in D^\# \mid \alpha \circ f \circ \gamma(y) \sqsubseteq y\} = \inf\{y \in D^\# \mid f \circ \gamma(y) \preceq \gamma(y)\}\,.$$

and thus for $D^+ = \gamma(D^\#)$ we get

$$\gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma) = \gamma(\inf\{y \in D^\# \mid f \circ \gamma(y) \preceq \gamma(y)\}) =$$

[since γ is a meet-morphism by Prop. 3.2 of [1]] $\inf\{\gamma(y) \in D \mid f \circ \gamma(y) \preceq \gamma(y)\}$

$$= \inf\{x \in D^+ \mid f(x) \preceq x\}\,.$$

1. Since

   $$(\gamma \circ \alpha \circ f)(\gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma)) = \gamma \circ (\alpha \circ f \circ \gamma) \circ \mathrm{lfp}(\alpha \circ f \circ \gamma) =$$
   $$[\text{definition of a fixpoint}]\ \gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma),$$

   $\gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma)$ is a fixpoint of $\gamma \circ \alpha \circ f$; thus $\gamma \circ \mathrm{lfp}(\alpha \circ f \circ \gamma)$ is greater than or equal to the least fixpoint of $\gamma \circ \alpha \circ f$.
2. Notice that $\mathrm{lfp}(\gamma \circ \alpha \circ f) = \gamma \circ \alpha \circ f \circ \mathrm{lfp}(\gamma \circ \alpha \circ f)$, thus in D^+. Moreover $f(\mathrm{lfp}(\gamma \circ \alpha \circ f)) \preceq$ [since $\gamma \circ \alpha$ is extensive] $\gamma \circ \alpha \circ f(\mathrm{lfp}(\gamma \circ \alpha \circ f)) =$ [definition of a fixpoint] $\mathrm{lfp}(\gamma \circ \alpha \circ f)$. So $\mathrm{lfp}(\gamma \circ \alpha \circ f)$ is in the set $\{x \in D^+ \mid f(x) \preceq x\}$ and hence greater than or equal to its infimum.
3. Notice that

   $$f \circ \gamma(\alpha \circ \mathrm{lfp}(\gamma \circ \alpha \circ f)) = \text{ [by definition of a fixpoint]}$$
   $$f \circ \gamma \circ \alpha \circ \gamma \circ \alpha \circ f(\mathrm{lfp}(\gamma \circ \alpha \circ f)) = [\gamma \circ \alpha \text{ is idempotent}]$$
   $$f \circ \gamma \circ \alpha \circ f(\mathrm{lfp}(\gamma \circ \alpha \circ f)) = [\text{definition of a fixpoint}]\ f(\mathrm{lfp}(\gamma \circ \alpha \circ f)) \preceq$$
   $$[\gamma \circ \alpha \text{ is extensive}]\ \gamma \circ \alpha \circ f(\mathrm{lfp}(\gamma \circ \alpha \circ f)) = [\text{definition of a fixpoint}]$$
   $$\mathrm{lfp}(\gamma \circ \alpha \circ f) \preceq [\gamma \circ \alpha \text{ is extensive}]\ \gamma(\alpha \circ \mathrm{lfp}(\gamma \circ \alpha \circ f)).$$

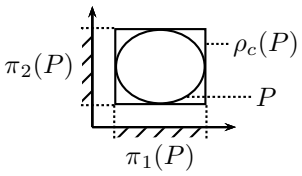
   So $\alpha \circ \mathrm{lfp}(\gamma \circ \alpha \circ f)$ is in the set $\{y \in D^\# \mid f \circ \gamma(y) \preceq \gamma(y)\}$, hence greater than or equal to its infimum.

□

1.9 Concrete Cartesian approximation

Up to now we identified the Flanagan-Qadeer model-checking as abstract fixpoint checking on an abstract domain. It turns out that the output of the Flanagan-Qadeer algorithm can also be characterized by a very simple approximation of multithreaded programs which is defined on the concrete domain. Now we define this approximation.
Recall that the *Cartesian approximation of a set of tuples* is the smallest Cartesian product containing this subset. Formally it is

$$\begin{gathered}\rho_c : \mathfrak{P}(X^n) \to \mathfrak{P}(X^n), \\ P \mapsto \{x \in X^n \mid \forall\, i \in \mathbb{N}_n \,\exists\, y \in P : y_i = x_i\}\,,\end{gathered}$$



where X is any set and $n \in \mathbb{N}^+$.

Proposition 1.9.1. Cartesian approximation is the approximation of the Cartesian Galois connection. Formally:

$$\rho_c = \gamma_c \circ \alpha_c\,.$$

Proof.

$$x \in \rho_c(P) \;\Leftrightarrow\; \forall\, i \in \mathbb{N}_n \,\exists\, y \in P : x_i = y_i \;\Leftrightarrow\; \forall\, i \in \mathbb{N}_n : x_i \in \pi_i(P) \;\Leftrightarrow$$
$$\forall\, i \in \mathbb{N}_n : x_i \in (\alpha_c(P))_i \;\Leftrightarrow\; x \in \prod_{i=1}^{n} (\alpha_c(P))_i \;\Leftrightarrow\; x \in \gamma_c \circ \alpha_c(P)\,.$$

□

An analog of Cartesian approximation on the concrete domain D is

$$\begin{gathered}\rho_{\mathrm{mc}} : D \to D \\ S \mapsto \{(g, l) \in \mathrm{Glob} \times \mathrm{Loc}^n \mid \forall\, i \in \mathbb{N}_n \,\exists\, (g, \bar{l}) \in S : \bar{l}_i = l_i\}\,.\end{gathered}$$

We call this map *multithreaded Cartesian approximation* of a set of program states since it simplifies to the Cartesian approximation of a set of tuples if Glob is a singleton.
It turns out that multithreaded Cartesian approximation is also representable in the abstract interpretation framework.

Proposition 1.9.2. Multithreaded Cartesian approximation is the approximation of the multithreaded Cartesian Galois connection. Formally:

$$\rho_{\mathrm{mc}} = \gamma_{\mathrm{mc}} \circ \alpha_{\mathrm{mc}}$$

Proof.

$$(g, l) \in \rho_{\mathrm{mc}}(S) \;\Leftrightarrow\; \forall\, i \in \mathbb{N}_n \,\exists\, (g, \bar{l}) \in S : \bar{l}_i = l_i \;\Leftrightarrow\; \forall\, i \in \mathbb{N}_n : (g, l_i) \in \pi_{0,i}(S)$$
$$\Leftrightarrow\; \forall\, i \in \mathbb{N}_n : (g, l_i) \in (\alpha_{\mathrm{mc}}(S))_i \;\Leftrightarrow\; (g, l) \in \gamma_{\mathrm{mc}} \circ \alpha_{\mathrm{mc}}(S)\,.$$

□

Instead of computing the least fixpoint of $\lambda y.\, \alpha_{\mathrm{mc}}(\mathrm{init} \cup \mathrm{post}(\gamma_{\mathrm{mc}}(y)))$, whose concretization represents a superset of the set of reachable states, we might as well compute the least fixpoint of $\lambda x.\, \rho_{\mathrm{mc}}(\mathrm{init} \cup \mathrm{post}(x))$ on the concrete domain.

For our tiny example from Section 1.4, the least fixed point computation on the concrete domain gives

$$\{(0, A, C), (0, A, D), (0, B, C), (0, B, D), (1, A, D), (1, B, D)\}$$

which coincides with $\gamma_{\mathrm{mc}}(\mathcal{R}_1, \mathcal{R}_2)$ computed by the Flanagan-Qadeer algorithm. By Thm. 1.8.2 the results will always coincide. Thus the Flanagan-Qadeer algorithm implements the computation of the program semantics with the multithreaded Cartesian approximation, which needs exponential time in the number of threads when implemented naively, in polynomial time.

1.10 The Owicki-Gries proof system

Now we describe the Owicki-Gries proof system from [15] excluding the proof rule for auxiliary variables. This proof system is a technique for annotating multithreaded programs for proving some (but not all) valid postconditions of multithreaded programs.

Each control location of each thread receives a formula speaking about shared variables and about local variables of the thread. A corresponding Hoare-style proof system for reasoning about annotation is supplied. If the program annotation is correct in that proof system, then the formulas at control locations hold for all time points in any run of the program. Especially it is possible to formulate and prove some (but not all) program invariants. The annotation size is linear in the number of threads, while the time to check the annotations is quadratic in the number of threads.

We adopt a very simple view of the Owicki-Gries proof system that allows us not to deal with the details of logic in which the annotation is written. Let a multithreaded program $(\mathrm{Glob}, \mathrm{Loc}, (\to_i)_{i=1}^n, \mathrm{init})$ be given such that

- $\mathrm{Loc} = \mathrm{LocData} \times \mathrm{PC}$ for some sets LocData and PC. Intuitively, PC is the set of program locations and LocData is the set of valuations of local data variables of the threads (here we slightly generalize the Owicki-Gries proof system, where the only syntactically local variable is the program counter).
- A set $C \subseteq \mathfrak{P}((\mathrm{Glob} \times \mathrm{Loc})^2)$ and two maps $\mathrm{srcloc}, \mathrm{trgloc} : C \to \mathrm{PC}$ exist such that such that for each $c \in C$ there is a thread $i \in \mathbb{N}_n$ such that

  $$c \subseteq \to_i \cap (\mathrm{Glob} \times \mathrm{LocData} \times \{\mathrm{srcloc}(c)\} \times \mathrm{Glob} \times \mathrm{LocData} \times \{\mathrm{trgloc}(c)\})$$

  and

  $$\bigcup_{i \in \mathbb{N}_n} \to_i \subseteq \bigcup C\,.$$

  Intuitively, C is the set of commands. Each command has exactly one source and one target location; statements of programming languages like `if b then S1 else S2` are modeled by two commands: one for the then-part, guarded by the condition `b`, the other for the else-part, guarded by the condition $\neg$`b`.

Definition 1.10.1. A *Hoare-triple* is a triple (P, c, Q) where $P, Q \subseteq \text{Glob} \times \text{LocData}$ and $c \in C$. We write $\{P\}c\{Q\}$ to denote the logical formula

$$\forall\, (g,(l,p),g',(l',p')) \in c :\ (g,l) \in P \Rightarrow (g',l') \in Q\,.$$

Definition 1.10.2. An *annotation* is a map $\text{ann} : \mathbb{N}_n \times \text{PC} \to \mathfrak{P}(\text{Glob} \times \text{LocData})$. Given a property $\text{safe} \subseteq \text{States}$, an *Owicki-Gries proof* of safe is an annotation ann such that the following conditions hold.

Sequential consistency: For each thread $i \in \mathbb{N}_n$, each command $c \in C$ and its control locations $p, p' \in \text{PC}$ with $p = \text{srcloc}(c)$ and $p' = \text{trgloc}(c)$ we have

$$\{\text{ann}(i,p)\}c\{\text{ann}(i,p')\}\,.$$

Noninterference: For each two different threads $i, j \in \mathbb{N}_n$ $(i \neq j)$, each control location $p \in \text{PC}$, each valuation of local variables $l \in \text{LocData}$ and each command $c \in C$ such that $c \subseteq \to_j$, for

$$G \;=\; \{g \mid (g,l) \in \text{ann}(i,p)\}$$

we have

$$\{(G \times \text{LocData}) \cap \text{ann}(j, \text{srcloc}(c))\}c\{G \times \text{LocData}\}\,.$$

Initial and final conditions: For sets of thread states

$$I_i := \{(g,(l,p)) \mid (g,l) \in \text{ann}(i,p)\} \qquad (i \in \mathbb{N}_n)$$

we have

$$\text{init} \subseteq \gamma_{\text{mc}}((I_i)_{i=1}^n) \subseteq \text{safe}\,.$$

1.11 Owicki-Gries = multithreaded Cartesian

We show that the Owicki-Gries proof system is able to prove the same properties as abstract fixpoint checking with multithreaded Cartesian abstraction and as thread-modular model-checking. The same result for an isomorphic Galois connection was stated in [4] without proof.

Theorem 1.11.1. An Owicki-Gries proof exists if and only if the multithreaded Cartesian proof exists. Formally: for each $\text{safe} \in D$ we have

$$\exists\, \text{Owicki-Gries proof of safe} \;\Leftrightarrow\; \gamma_{\text{mc}}(\text{lfp}(\lambda y.\, \alpha_{\text{mc}}(\text{init} \cup \text{post}(\gamma_{\text{mc}}(y))))) \subseteq \text{safe}\,.$$

Idea. Given an Owicki-Gries proof, show that it denotes an inductive invariant (i.e. a set which contains the initial states and is closed under post) expressible in the abstract domain. Given the least fixpoint of the abstract fixpoint checking operator, construct the annotation from its components.

Proof.

"⇒": Assume an Owicki-Gries proof ann. Let $I := \gamma_{\text{mc}}((I_i)_{i=1}^n)$. Notice that

$$\begin{aligned}
&\gamma_{\text{mc}}(\text{lfp}(\lambda x.\, \alpha_{\text{mc}}(\text{init} \cup \text{post}(\gamma_{\text{mc}}(x))))) \text{ [Tarski's fixpoint theorem]}\\
=&\gamma_{\text{mc}}(\text{inf}_{D^\#}\{y \mid \alpha_{\text{mc}}(\text{init} \cup \text{post}(\gamma_{\text{mc}}(y))) \sqsubseteq y\}) \text{ [}\gamma_{\text{mc}}\text{ is meet-morphism]}\\
=&\,\text{inf}_D\{\gamma_{\text{mc}}(y) \mid \alpha_{\text{mc}}(\text{init} \cup \text{post}(\gamma_{\text{mc}}(y))) \sqsubseteq y\} \text{ [def. of Galois conn.]}\\
=&\,\text{inf}_D\{\gamma_{\text{mc}}(y) \mid \text{init} \cup \text{post}(\gamma_{\text{mc}}(y))) \subseteq \gamma_{\text{mc}}(y)\}\\
=&\,\text{inf}_D\{x \in \gamma_{\text{mc}}(D^\#) \mid \text{init} \cup \text{post}(x) \subseteq x\}\,. \qquad (1.3)
\end{aligned}$$

We know that $I \subseteq \text{safe}$. To show that the above infimum (1.3) is included in safe, it suffices to show that I belongs to the set of states over which the infimum is taken.

$I \in \gamma_{\text{mc}}(D^{\#})$: True by definition of I.

$\text{init} \subseteq I$: True by the initial condition.

$\text{post}(I) \subseteq I$: Consider a state $(g, (l_i, p_i)_{i=1}^n) \in I$ and its any successor $(g', (l'_i, p'_i)_{i=1}^n)$ due to a step of a thread $k \in \mathbb{N}_n$. There is some command $c \in C$ such that $(g, (l_k, p_k), g', (l'_k, p'_k)) \in c \subseteq \to_k$. Since $(g, (l_k, p_k)) \in I_k$, we have $(g, l_k) \in \text{ann}(k, p_k)$. By sequential consistency we get $(g', l'_k) \in \text{ann}(k, p'_k)$ and so $(g', (l'_k, p'_k)) \in I_k$. Consider any other thread $j \in \mathbb{N}_n \setminus \{k\}$. Let $G = \{g \mid (g, l_j) \in \text{ann}(j, p_j)\}$. We have $(g, (l_j, p_j)) \in I_j$, so $(g, l_j) \in \text{ann}(j, p_j)$, so $g \in G$. Thus $(g, l_k) \in G \times \text{LocData} \cap \text{ann}(k, p_k)$. By noninterference we have $(g', l'_k) \in G \times \text{LocData}$, thus $g' \in G$, so $(g', l_j) \in \text{ann}(j, p_j)$ and $(g', (l'_j, p'_j)) = (g', (l_j, p_j)) \in I_j$. So for any $i \in \mathbb{N}_n$ (both for $i = k$ and $i \neq k$) we get $(g', (l'_i, p'_i)) \in I_i$. Thus $(g', (l'_i, p'_i)_{i=1}^n) \in I$.

"$\Leftarrow$": Let the $\gamma_{\text{mc}}(\text{lfp}(\lambda y.\, \alpha_{\text{mc}}(\text{init} \cup \text{post} \circ \gamma_{\text{mc}}(y)))) \subseteq \text{safe}$. Thm. 1.7.2 and Prop. 1.7.1 imply $\gamma_{\text{mc}}((\mathcal{R}_i)_{i=1}^n) \subseteq \text{safe}$ for the least fixpoint $(\mathcal{R}_i)_{i=1}^n$ of the rule set FQ'. Define an annotation function by $\text{ann}(i, p) := \{(g, l) \mid (g, (l, p)) \in \mathcal{R}_i\}$. Now we show that ann is an Owicki-Gries proof.

Sequential consistency: Let $c \in C$, $i \in \mathbb{N}_n$, $p, p' \in \text{PC}$, $p = \text{srcloc}(c)$, $p' = \text{trgloc}(c)$ and $c \subseteq \to_i$. Let $(g, (l, p), g', (l', p')) \in c$ and $(g, l) \in \text{ann}(i, p)$. Then $(g, (l, p)) \in \mathcal{R}_i$. Since $c \subseteq \to_i$, the STEP' rule implies $(g', (l', p')) \in \mathcal{R}_i$. Thus $(g', l') \in \text{ann}(i, p')$.

Noninterference: Let $i, j \in \mathbb{N}_n$, $p \in \text{PC}$, $l \in \text{LocData}$, $c \in C$ with $i \neq j$, $c \subseteq \to_j$ and $G = \{g \mid (g, l) \in \text{ann}(i, p)\}$. Let $(g, (\bar{l}, \bar{p}), (g', (\bar{l}', \bar{p}'))) \in c$ and $(g, \bar{l}) \in G \times \text{LocData} \cap \text{ann}(j, \bar{p})$. Since $g \in G$, we have $(g, l) \in \text{ann}(i, p)$, thus $(g, (l, p)) \in \mathcal{R}_i$. By the ENV' rule, applied to the premises $(g, (l, p)) \in \mathcal{R}_i$, $(g, (\bar{l}, \bar{p})) \in \mathcal{R}_j$, $(g, (\bar{l}, \bar{p})) \to_j (g', (\bar{l}', \bar{p}'))$ and $i \neq j$, we obtain $(g', (l, p)) \in \mathcal{R}_i$, thus $(g', l) \in \text{ann}(i, p)$, so $g' \in G$ and $(g', \bar{l}) \in G \times \text{LocData}$.

Initial and final condition: Notice that $I_i = \mathcal{R}_i$ for all $i \in \mathbb{N}_n$. Thus $\text{init} \subseteq \gamma_{\text{mc}} \circ \alpha_{\text{mc}}(\text{init}) \subseteq \gamma_{\text{mc}}(\text{lfp}(\lambda y.\, \alpha_{\text{mc}}(\text{init}) \sqcup \alpha_{\text{mc}} \circ \text{post} \circ \gamma_{\text{mc}}(y))) = \underbrace{\gamma_{\text{mc}}((\mathcal{R}_i)_{i=1}^n)}_{\gamma_{\text{mc}}((I_i)_{i=1}^n)} \subseteq \text{safe}$.

□

The theorem has an interesting consequence:

Corollary 1.11.2. There exists an algorithm that, given a program that is known to have an Owicki-Gries proof, finds this proof in polynomial time in the program size.

Proof. The algorithm takes a finite-state multithreaded program and runs abstract fixpoint computation with multithreaded Cartesian abstraction, resulting in the least fixpoint $(I_i)_{i=1}^n$, then it constructs an annotation as in the proof of Thm. 1.11.1, part "$\Leftarrow$". By Thm. 1.11.1, part "$\Rightarrow$", the concretization of $(I_i)_{i=1}^n$ proves the property. By Thm. 1.7.2, $(I_i)_{i=1}^n$ is the least fixpoint of FQ'. By the proof of Thm. 1.11.1, part "$\Leftarrow$", the annotation is an Owicki-Gries proof. By Prop. 1.5.4 the number of iterations is bounded by $\mathfrak{h}(D^{\#}, \sqsubseteq)$, which is polynomial in the program size. □

1.12 Boundary of the Flanagan-Qadeer algorithm

Now we try to push the Flanagan-Qadeer algorithm to increase precision without losing polynomial complexity.

A usual way to change a search algorithm is to use "frontier search", which forgets the nodes of the search graph that were expanded in the past. The idea is to apply multithreaded Cartesian approximation not to all of the discovered states, but only to the latest discovered states:

$$T^{(0)} = \alpha_{\mathrm{mc}}(\mathrm{init}) \quad \text{and} \quad T^{(i+1)} = \mathrm{post}^{\#}_{\mathrm{mc}}(T^{(i)}) \qquad (i \geq 0)\,,$$

where $\mathrm{post}^{\#}_{\mathrm{mc}} = \alpha_{\mathrm{mc}} \circ \mathrm{post} \circ \gamma_{\mathrm{mc}}$. The sequence stops when $\gamma_{\mathrm{mc}}(T^{(k+1)}) \subseteq \bigcup_{i=0}^{k} \gamma_{\mathrm{mc}}(T^{(i)})$. For this k one can show that $\bigcup_{i=0}^{k} \gamma_{\mathrm{mc}}(T^{(i)})$ is an inductive invariant.

These iterates are computed by the following rules:

$$\textsc{InitF}\ \frac{}{\mathrm{init}_i \subseteq T_i^{(0)}}\ i \in \mathbb{N}_n$$

$$\textsc{StepF}_{ij}\ \frac{(g, l_i) \in \mathcal{R}_i \qquad (g, l_j) \in \mathcal{R}_j \qquad (g, l_j) \to_j (g', l'_j)}{(g', l_i) \in \mathcal{R}'_i \qquad (g', l'_j) \in \mathcal{R}'_j}\ i \neq j \wedge i, j \in \mathbb{N}_n.$$

Except for the primed versions $\mathcal{R}'_i$ and $\mathcal{R}'_j$ in the conclusion of the rule, this is the same as the combination of STEP' and ENV' rules from the rule system FQ'. If $T^{(i)} = (\mathcal{R}_1, \ldots, \mathcal{R}_n)$ is the i^{th} element of the iteration sequence then $T^{(i+1)} = (\mathcal{R}'_1, \ldots, \mathcal{R}'_n)$ is computed by the above rule. So each step of the new iteration scheme is polynomial.

Multithreaded Cartesian abstraction is not applied to all states but only to the latest discovered, thus we might expect the new iteration scheme to be more precise. Moreover, each step of the new iteration scheme is polynomial. However, it turns out that the number of steps might grow too large:

Theorem 1.12.1. Frontier search with Cartesian abstraction has exponential worst-case runtime in the number of threads.

Proof. Consider the following implementation of a binary counter.

Example 1.12.2. Below you see a program with $n \geq 2$ threads whose single run has length at least 2^n. The program has $n + 1$ global states, each thread has two local states. The statements in brackets <> are atomic.

Global variable with initial value:
$t = 1$ (takes values from $\{0, ..., n\}$)
Thread 1**:**
`0:` `wait until` $t = 1$;
`1:` $<t := 2;$ `goto 0;`$>$
Thread i $(1 < i < n)$**:**
`0:` $<$`wait until` $t = i;$ $t := 1;>$
`1:` $<$`wait until` $t = i;$ $t := i + 1;$ `goto 0;`$>$
Thread n**:**
`0:` $<$`wait until` $t = n;$ $t := 1;>$
`1:` $<$`wait until` $t = n;$ $t := 0;$ `goto 0;`$>$
The program performs repeated increment using a carry. The local state of the i^{th} thread represents the position $i - 1$ of the number $(1 \leq i \leq n)$. The carry

position is stored in the global variable t. The value $t = 0$ means the carry is nowhere.

Below is the single run for $n = 3$ where pc_i is the program counter of the i^{th} thread. Each column represents a state of the whole program, a successor state is to the right of its predecessor:

variable	*	*		*	*			*	*		*	*			
t	1	1	2	1	1	2	3	1	1	2	1	1	2	3	0
pc_1	0	1	0	0	1	0	0	0	1	0	0	1	0	0	0
pc_2	0	0	0	1	1	1	0	0	0	0	1	1	1	0	0
pc_3	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0

Let us consider the columns marked by the star (*), i.e the columns with $t = 1$ (the carry is above the 0^{th} position). One sees that the values of the program counters (pc_3, pc_2, pc_1) evolve like a binary counter. The formal inductive proof that this program implements the binary counter and that it has exactly one run is left as an exercise for the reader.

The size of the program description, i.e. the size of the sets Loc and Glob, is polynomial in n, while the run is exponential in n. Let's examine the sets $T^{(m)}$ of the frontier search with multithreaded Cartesian abstraction for this program.

Notice that all components of $T^{(0)}$ are the singletons $\{(1, 0)\}$. If for some $m \geq 0$ all components of the tuple $T^{(m)}$ contain at most one element each, then $\gamma_{\text{mc}} T^{(m)}$ contains at most one element, and hence $\text{post} \circ \gamma_{\text{mc}}(T^{(m)})$ has at most one element, so $T^{(m+1)} = \alpha_{\text{mc}} \circ \text{post} \circ \gamma_{\text{mc}}(T^{(m)})$ is a tuple of sets which are either all empty sets or all singletons. Since $\gamma_{\text{mc}}(T^{(m)})$ contains $\text{post}^m(\text{init})$, we inductively follow that $\gamma_{\text{mc}}(T^{(m)}) = \text{post}^m(\text{init})$ for all $m \geq 0$, i.e. no precision gets lost. Especially the sequence $(T^{(m)})_{0 \leq m \leq k}$ is exponentially long. □

So frontier search is the precision limit of the multithreaded Cartesian abstraction.

Remark that the set of reachable states of binary counter in Example 1.12.2 is so big that the output of the Flanagan-Qadeer algorithm is exact:

$$\begin{aligned}
\mathcal{R}_1 = \quad & \{(1,0),(1,1),(2,0), && (3,0), && (0,0)\}, \\
\mathcal{R}_2 = \quad & \{(1,0),(1,1),(2,0),(2,1), && (3,0), && (0,0)\}, \\
\mathcal{R}_3 = \quad & \{(1,0),(1,1),(2,0),(2,1), && (3,0),(3,1), && (0,0)\}.
\end{aligned}$$

Namely, the concretization of the output $\gamma_{\text{mc}}(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3)$ gives exactly the set of reachable states. So this is also an example of when an attempt of regaining precision by frontier search doesn't increase precision but requires exponential time.

The binary counter example has a property that the number of global states grows linearly with the number of threads. Can one get states reachable only by exponentially long runs with a sublinear or even constant number of global states? This would give an answer to the question whether exponentially long runs occur only due to the growing number of threads or need a growing shared part. We pose the following

Problem 1.12.3. Prove or give a counterexample. There is no family $(P_n)_{n \geq 1}$ of multithreaded programs so that

- the n^{th} program P_n consists of n threads;
- the number of the global and local states is bounded by a constant which is independent on n;
- the diameter of the transition graph of the n^{th} program P_n is asymptotically exponential in n, i.e. there is a constant $c > 1$ so that for almost all $n \in \mathbb{N}$ the diameter of the transition graph of P_n exceeds c^n.

We do not see how to solve this problem at the moment.

1.13 Statically regaining precision

The inherent problem of the original Flanagan-Qadeer algorithm is that its abstraction is too coarse. It forgets much of the dependencies between the threads. We show an old and discover a new technique that overcomes some of the precision loss and give a practical polynomial-time algorithm based on it.

Loosely speaking, multithreaded Cartesian approximation "forgets" temporal properties of the program: a state of one thread that occurs late in some computation can be combined with a state of another thread that occurs early in another computation.

Example 1.13.1. In this example, the Flanagan-Qadeer algorithm cannot prove that the first thread never reaches label D:

Global variable g=0

```
Thread 1:            ‖ Thread 2:
A: wait until g=1;   ‖ E: g:=1;
B: wait until g=0;   ‖ F: g:=0;
C: wait until g=1;   ‖ G:
D:                   ‖
```

The Flanagan-Qadeer algorithm computes the sets

$$\begin{aligned}\mathcal{R}_1 =&\{(0,A),(0,B),(0,C),(0,D),\\ &(1,A),(1,B),(1,C),(1,D)\}\,,\\ \mathcal{R}_2 =&\{(0,E),(0,G),(1,F)\}\,.\end{aligned}$$

Since the label D is discovered by $(0,D) \in \mathcal{R}_1$, the property was not proven.

Multithreaded Cartesian approximation also loses precision by "forgetting" dependencies between threads even in successors of a single program state. For instance, the Flanagan-Qadeer algorithm is unable to prove mutual exclusion with locks (see example in [5]) of the following program.

Example 1.13.2. In the program below, execution never reaches the state $pc_1 = pc_2 = B$.

Global variable m=0

```
Thread 1:              ‖ Thread 2:
A: <wait until m=0;    ‖ A: <wait until m=0;
     m:=1;>            ‖      m:=1;>
B:                     ‖ B:
```

The Flanagan-Qadeer algorithm computes the sets $\mathcal{R}_1 = \mathcal{R}_2 = \{(0,A), (1,A),(1,B)\}$, whose concretization $\gamma_{\text{mc}}(\mathcal{R}_1,\mathcal{R}_2)$ contains $(1,B,B)$, which violates mutual exclusion.

Now we give two techniques that solve the above problems while still being polynomial in time.

1.13.1 Exposing Local Information

A standard technique for regaining precision of thread-modular algorithms is to expose some of the private state to the correctness proofs of the other threads. For instance, for the Owicki-Gries verification method [15], write-only shared variables are added which expose some information about the threads' local state. These variables are called auxiliary or ghost variables. By exposing all information about the local state it is possible to get completeness at the cost of verification time. The Flanagan-Qadeer algorithm is subjected to the same improvement.

For the mutual exclusion Example 1.13.2 it is possible to choose the variables in several ways. One way is to introduce a variable $t \in \{1, 2\}$, which is set to 1 by every transition of the first thread and to 2 by every transition of the second thread. After doing this, the critical thread state $pc_1 = B \wedge m = 1 \wedge t = 1$ of the first thread is not mixed with the critical thread state $pc_2 = B \wedge m = 1 \wedge t = 2$ of the second thread: Multithreaded Cartesian abstraction will treat the different shared parts separately and would not produce a state satisfying $pc_1 = pc_2 = B$. The "quadratic thread-modular method", discovered by the author in [13, 12], is just a hidden way of making the thread number of a transition a shared variable.

1.13.2 Relaxed Frontier Search

Frontier search is a boundary of the thread-modular method: it has exponential worst-case runtime. However, one can combine the Flanagan-Qadeer algorithm and the frontier search such that the new method has polynomial runtime.

Definition 1.13.3 (Relaxed Frontier Sequence)**.** Let the sequence $(T^{(i)})_{i\geq 0}$ of elements of $D^\#$ be defined recursively by

$$T^{(0)} = \alpha_{\mathrm{mc}}(\mathrm{init})\,,$$
$$T^{(i+1)} = \begin{cases} \mathrm{post}^\#_{\mathrm{mc}}(T^{(i)})\,, & \text{if } \mathrm{post}^\#_{\mathrm{mc}}(T^{(i)}) \not\sqsubseteq \sqcup_{j=0}^{i} T^{(j)}, \\ T^{(i)} \sqcup \mathrm{post}^\#_{\mathrm{mc}}(T^{(i)})\,, & \text{if } \mathrm{post}^\#_{\mathrm{mc}}(T^{(i)}) \sqsubseteq \sqcup_{j=0}^{i} T^{(j)} \end{cases} \qquad (i \geq 0).$$

Let $k = \min\{i \mid T^{(i+1)} \sqsubseteq T^{(i)}\} \in \mathbb{N}_0 \dot\cup \{\infty\}$ (where $\infty = \min \emptyset$).

This (in general non-monotone) sequence is the basis of the new algorithm, which needs only elements of the sequence up to position k. First we prove that k is well-defined, and is at most quadratic in the number of threads. Let $g = |\mathrm{Glob}|$ the number of the shared states and $l = |\mathrm{Loc}|$ the number of the local states.

Proposition 1.13.4. On finite-state multithreaded programs the relaxed frontier method needs at most $k \leq (gln)(gln + 1) = O((gln)^2)$ iterations.

Idea. At each position either the current iterate increases or the join of all known iterates increases. The join might increase as often as the height of the abstract lattice, and between each two such neighboring increases there might be a sequence of at most height-of-the-abstract-lattice steps.

Proof. First we prove that k is finite. The set $D^\#$ is finite, so the monotone sequence $(\sqcup_{i=0}^{k'} T^{(i)})_{k' \geq 0}$ stabilizes for some $k' \geq 0$, i.e. $T^{(j)} \sqsubseteq \sqcup_{i=0}^{k'} T^{(i)}$ for all $j \geq k'$. Take the smallest such k'. For all $j \geq k'$ we have $\mathrm{post}^\#_{\mathrm{mc}}(T^{(j)}) \sqsubseteq T^{(j+1)} \sqsubseteq \sqcup_{i=0}^{k'} T^{(i)} = \sqcup_{i=0}^{j} T^{(i)}$. By definition of the algorithm, the sequence $(T^{(j)})_{j \geq k'}$ is monotonically increasing. The domain is finite, so there is a $k \geq k'$ with $T^{(k+1)} = T^{(k)}$.

Now we derive the number of iteration steps. Notice that for any $j < k$, either $T^{(j)} \sqsubset T^{(j+1)}$ or $\sqcup_{i=0}^{j} T^{(i)} \sqsubset \sqcup_{i=0}^{j+1} T^{(i)}$. The set $D^\#$ has chains of length at most $gln+1$ by Prop. 1.5.4, so in the increasing sequence $(\sqcup_{i=0}^{j} T^{(i)})_{j \geq 0}$ there are at most gln positions with a strict increase. Take any "neighbour" positions $a < b$ with $a > 0$ and

$$\sqcup_{i=0}^{a-1} T^{(i)} \neq \sqcup_{i=0}^{a} T^{(i)} \text{ and } \sqcup_{i=0}^{b-1} T^{(i)} \neq \sqcup_{i=0}^{b} T^{(i)}$$

so that for all c with $a < c < b$ we have

$$\sqcup_{i=0}^{c-1} T^{(i)} = \sqcup_{i=0}^{c} T^{(i)} \,.$$

The sequence $(T^{(c)})_{a \leq c < b}$ is increasing by the definition of the relaxed frontier sequence and thus $b - a \leq \mathfrak{h}(D^\#, \sqsubseteq) = gln + 1$. Finally $k - k' \leq gln$. Since the number of positions with a strict increase of $\sqcup_{i=0}^{j} T^{(i)}$ is at most gln, and for each such position there is a following chain $(T^{(c)})_c$ with at most $gln + 1$ elements including the element on this position, $k \leq (gln)(gln + 1)$. □

Countably many iterations suffice to get a superset of states reachable from the initial ones because of

Proposition 1.13.5. The set $\bigcup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}}(T^{(i)})$ is an inductive invariant.

Proof. By extensivity of $\gamma_{\mathrm{mc}} \circ \alpha_{\mathrm{mc}}$ we obtain $\mathrm{init} \subseteq \gamma_{\mathrm{mc}} \circ \alpha_{\mathrm{mc}}(\mathrm{init}) = \gamma_{\mathrm{mc}}(T^{(0)})$. Moreover

$$\mathrm{post}(\cup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}}(T^{(i)})) = \bigcup_{i \in \mathbb{N}_0} \mathrm{post} \circ \gamma_{\mathrm{mc}}(T^{(i)}) \subseteq$$

$$\bigcup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}} \circ \alpha_{\mathrm{mc}} \circ \mathrm{post} \circ \gamma_{\mathrm{mc}}(T^{(i)}) = \bigcup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}} \circ \mathrm{post}^\#_{\mathrm{mc}}(T^{(i)}) \subseteq \bigcup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}}(T^{(i+1)})$$

$$= \bigcup_{i \in \mathbb{N}^+} \gamma_{\mathrm{mc}}(T^{(i)}) \subseteq \bigcup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}}(T^{(i)}) \,.$$

□

Proposition 1.13.6. For finite-state programs $\bigcup_{i=0}^{k} \gamma_{\mathrm{mc}}(T^{(i)})$ is an inductive invariant.

Proof. By definition of k we have $T^{(k+1)} \sqsubseteq T^{(k)}$. If for some $j > k$ we have $T^{(j)} \sqsubseteq T^{(k)}$, then $T^{(j+1)} \sqsubseteq T^{(j)} \sqcup \mathrm{post}^\#_{\mathrm{mc}}(T^{(j)}) \sqsubseteq$ [by assumption and monotonicity of $\mathrm{post}^\#_{\mathrm{mc}}$] $T^{(k)} \sqcup \mathrm{post}^\#_{\mathrm{mc}}(T^{(k)}) \sqsubseteq T^{(k)} \sqcup T^{(k+1)} \sqsubseteq$ [by definition of k] $T^{(k)}$. By induction $T^{(j)} \sqsubseteq T^{(k)}$ for all $j \geq k$. Thus $\gamma_{\mathrm{mc}}(T^{(j)}) \subseteq \gamma_{\mathrm{mc}}(T^{(k)})$ for all $j \geq k$, implying $\bigcup_{i=0}^{k} \gamma_{\mathrm{mc}}(T^{(i)}) = \bigcup_{i \in \mathbb{N}_0} \gamma_{\mathrm{mc}}(T^{(i)})$. □

Algorithm 1 implements abstract fixpoint model checking with relaxed frontier search in linear space in quadratic number of iterations.

Input: Initial states init $\subseteq$ States, error state $f \in$ States
Output: "safe" or "don't know"
$T := \bot$;
$T' := \alpha_{\mathrm{mc}}(\mathrm{init})$;
$All := T'$;
while $T' \not\sqsubseteq T$ *and* $\alpha_{\mathrm{mc}}(\{f\}) \not\sqsubseteq T'$ **do**
  $T := T'$;
  $P := \mathrm{post}^{\#}_{\mathrm{mc}}(T)$;
  if $P \sqsubseteq All$ **then**
    $T' := T \sqcup P$;
  else
    $T' := P$;
  end
  $All := All \sqcup T'$;
end
if $\alpha_{\mathrm{mc}}(\{f\}) \sqsubseteq T'$ **then**
  return *"don't know"*;
else
  return *"safe"*;
end

Algorithm 1: Relaxed Frontier Search with multithreaded Cartesian abstraction regains precision in polynomial time

The map $\mathrm{post}^{\#}_{\mathrm{mc}}$ can be computed by the StepF rule and the number of iterations is quadratic in the number of threads. The used space is dominated by the size of All, which can be stored in $O(n|\mathrm{Glob}||\mathrm{Loc}|)$ space, which is linear in the number of threads.

Let's look at what the frontier search does to the program from Example 1.13.1. Below is the relaxed frontier sequence:

i	elements of the first component of $T^{(i)}$	elements of the second component of $T^{(i)}$
0	$(0, A)$	$(0, E)$
1	$(1, A)$	$(1, F)$
2	$(1, B)$, $(0, A)$	$(1, F)$, $(0, G)$
3	$(0, B)$	$(0, G)$
4	$(0, C)$	$(0, G)$

The elements after position $k = 4$ are all $((0, C), (0, G))$.

The thread states of the form $(\_, D)$ are not in $T^{(i)}$ for any i, so the property is proven. The computation on this special example was equivalent to that of the pure frontier search, since each internal step a new thread state was discovered. This will be the case for linear loop-free programs.

For the binary counter example 1.12.2, the computed invariant is equal to that of the pure Flanagan-Qadeer method, since the latter contains exactly the reachable states. However, the number of iterations is greater.

Chapter 2

Refined Thread-Modular Reasoning

In this chapter we are going to tune the precision of thread-modular reasoning. First we will explain and discuss the tuning principle on an arbitrary Galois connection in Sec. 2.1. In Sec. 2.2 we will apply this principle to the Cartesian Galois connection and its multithreaded variant; in particular we'll discuss the complexity and introduce algorithms for the least fixpoint computation.

2.1 Exceptions

Sometimes a Galois connection is too coarse to prove the desired property of an analyzed program. In view of Thm. 1.8.2, the loss of precision occurs when in a fixpoint iteration a set of states X is approximated by $\rho(X)$. One possibility is to move to a new abstract domain with a new Galois connection which is more precise than the current Galois connection. However, it was not clear how to refine a Galois connection while using the same abstract domain. We are filling this gap, presenting a technique for refining a large class of Galois connections, namely for those with a zero-strict concretization function, i.e. a concretization function that maps the lowest element of the abstract domain to the lowest element of the concrete domain.

Throughout this section, assume that $(D, \subseteq)$ is a complemented distributed complete lattice. By the Stone's representation theorem for Boolean algebras, it is isomorphic to a field of sets, moreover the complement of any element is unique. We denote by x^c the complement of x. Further let $(D^\#, \sqsubseteq)$ be a complete lattice and (α, γ) a Galois connection between $(D, \subseteq)$ and $(D^\#, \sqsubseteq)$ such that γ is zero-strict. We denote the least element of the abstract lattice by $\bot = \inf(D^\#)$. Moreover, let $D^{\#+} = \alpha(D)$.

The idea of parameterizing the Galois connection is introducing an element $E \in D$ (called the Exception Element), which, intuitively, should not be subjected to abstraction. Formally, consider the *exceptional abstraction* and *exceptional concretization* maps

$$\alpha_E : D \to D, \quad X \mapsto X \cap E^c$$

and

$$\gamma_E : D \to D, \quad X \mapsto X \cup E\,.$$

Proposition 2.1.1. The pair (α_E, γ_E) is a Galois connection. Formally:

$$\forall\, X, Y \in D: \quad \alpha_E(X) \subseteq Y \Leftrightarrow X \subseteq \gamma_E(Y)\,.$$

Proof. Let $X, Y \in D$.

"$\Rightarrow$": Let $\alpha_E(X) \subseteq Y$, i.e. $X \cap E^c \subseteq Y$. Then $X \subseteq X \cup E = (X \cup E) \cap \sup(D)$ $= (X \cup E) \cap (E^c \cup E) =$ [distributivity] $(X \cap E^c) \cup E \subseteq$ [since $X \cap E^c \subseteq Y$] $Y \cup E = \gamma_E(Y)$.

"$\Leftarrow$": Let $X \subseteq \gamma_E(Y)$, i.e. $X \subseteq Y \cup E$. Then $\alpha_E(X) = X \cap E^c \subseteq$ [since $X \subseteq Y \cup E$] $(Y \cup E) \cap E^c =$ [distributivity] $(Y \cap E^c) \cup \underbrace{(E \cap E^c)}_{\inf(D)} = Y \cap E^c \subseteq Y$. $\square$

The composition $(\alpha \circ \alpha_E, \gamma_E \circ \gamma)$ of Galois connections is a Galois connection, the so-called *parameterized* Galois connection. Let $\text{init} \in D$ be any element and $\text{post} : D \to D$ monotone, usually the successor map of the analyzed program. The parameterized analysis computes $\text{lfp}(\lambda Y.\, \alpha \circ \alpha_E(\text{init} \cup \text{post} \circ \gamma_E \circ \gamma(Y)))$. The concretization of this least fixpoint

$$\gamma_E \circ \gamma(\text{lfp}\,(\lambda Y.\, \alpha \circ \alpha_E(\text{init} \cup \text{post} \circ \gamma_E \circ \gamma(Y))))$$

overapproximates the least fixpoint of $\lambda x.\, \text{init} \cup \text{post}(x)$ by a standard result of the abstract interpretation framework (cf. [1], Prop. 13), so soundness of the analysis is guaranteed. Choosing E as its any postfixpoint makes the above concretization even equal to this postfixpoint. So it is even possible to get exactly the least fixed point of $\lambda x.\, \text{init} \cup \text{post}(x)$.

Readers who are not interested in tuning general Galois connections might want to continue with the next section 2.2.

2.1.1 Varying precision

Now we discuss in detail how the choice of the Exception Element influences precision of the abstract fixpoint model checking when using the parameterized Galois connection.

If $E = \emptyset$, then the exceptional abstraction and concretization maps are identities. There is no difference, compared to the original Galois connection.

Now let $f : D \to D$ be monotone. For example, f can be $\lambda x.\, \text{init} \cup \text{post}(x)$ for the set of initial states init and successor map post.

Proposition 2.1.2. An element is a postfixpoint of f if and only if the parameterized concretization of the abstract fixpoint (with Galois connection parameterized by this element) gives exactly this element. Formally:

$$\forall\, E \in D: \;\; f(E) \subseteq E \;\Leftrightarrow\; \gamma_E \circ \gamma(\text{lfp}(\alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma)) = E\,.$$

Proof. Let $E \in D$.

"$\Rightarrow$": Let $f(E) \subseteq E$. From the zero-strictness of the concretization map, we have $\gamma_E \circ \gamma(\bot) = E$ and by postfixpoint property of E we have $f \circ \gamma_E \circ \gamma(\bot) \subseteq E$, and thus $\alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma(\bot) = \bot$ (that α maps bottom to bottom follows from the definition of a Galois connection). So the abstract fixpoint is

$\mathrm{lfp}(\alpha\circ\alpha_E\circ f\circ\gamma_E\gamma) = \bot$, thus its concretization is $\gamma_E\circ\gamma\circ\mathrm{lfp}(\alpha\circ\alpha_E\circ f\gamma_E\circ\gamma) = E$.
"$\Leftarrow$": True for all Galois connections and not only for $(\alpha\circ\alpha_E, \gamma_E\circ\gamma)$. So let $(\bar{\alpha},\bar{\gamma})$ be a Galois connection between the concrete domain D and the abstract domain $D^\#$ and $E = \bar{\gamma}\circ\mathrm{lfp}(\bar{\alpha}\circ f\circ\bar{\gamma})$. Then $f(E) = f\circ\bar{\gamma}\circ\mathrm{lfp}(\bar{\alpha}\circ f\circ\bar{\gamma})$. The map $\bar{\gamma}\circ\bar{\alpha}$ is extensive, so $f(E) \subseteq (\bar{\gamma}\circ\bar{\alpha})\circ f\circ\bar{\gamma}\circ\mathrm{lfp}(\bar{\alpha}\circ f\circ\bar{\gamma}) = \bar{\gamma}\circ(\bar{\alpha}\circ f\circ\bar{\gamma})\circ\mathrm{lfp}(\bar{\alpha}\circ f\circ\bar{\gamma}) = \bar{\gamma}\circ\mathrm{lfp}(\bar{\alpha}\circ f\circ\bar{\gamma}) = E$. □

For program analysis this means: if parameterizing the Galois connection by a set gives the set back, then this set is an inductive invariant of the program. Vice versa, if a set is an inductive invariant, then parameterizing the Galois connection by this set gives this set back, so we neither gain nor lose precision.

Since the least fixpoint of f exists and is a postfixpoint, we obtain

Corollary 2.1.3. There is an exception set so that abstract fixpoint computation with the Galois connection parameterized by this set gives the exact answer. Formally:

$$\exists\, E:\ \gamma_E\circ\gamma\circ\mathrm{lfp}(\alpha\circ\alpha_E\circ f\circ\gamma_E\circ\gamma) = \mathrm{lfp}(f)\,.$$

For program analysis this means that the parameterized analysis is complete in the sense that there is a choice of the exception set that leads to the best inductive invariant.

Corollary 2.1.4. An element $A \in D$ is an abstract postfixpoint if and only if the abstract fixpoint computation with Galois connection parameterized by the concretization of A returns the concretization of A. Formally:

$$\forall\, A \in D^\#:\quad \alpha\circ f\circ\gamma(A) \sqsubseteq A \ \Leftrightarrow\ \gamma_{\gamma(A)}\circ\gamma\circ\mathrm{lfp}(\alpha\circ\alpha_{\gamma(A)}\circ f\circ\gamma_{\gamma(A)}\circ\gamma) = \gamma(A)\,.$$

Proof. Since (α,γ) is a Galois connection, $\alpha\circ f\circ\gamma(A) \sqsubseteq A \Leftrightarrow f\circ\gamma(A) \subseteq \gamma(A)$. The result follows with $E = \gamma(A)$ from Prop. 2.1.2. □

For program analysis this means: whenever A is an abstract inductive invariant, the abstract fixpoint analysis with parameter $\gamma(A)$ gives the parameter itself. Vice versa, if the abstract fixpoint analysis with Galois connection parameterized by $\gamma(A)$ gives $\gamma(A)$ back, then it is an abstract inductive invariant. Especially one cannot increase precision by taking the concretization of the least abstract fixpoint $\gamma\circ\mathrm{lfp}(\alpha\circ f\circ\gamma)$ as an exception set.

Now we show how the fixpoints are related when the parameter increases or decreases.

Proposition 2.1.5. Let A be any concrete element and X be a result of abstract fixpoint computation with abstraction parameterized by A. Then taking as an exception set any element between A and the parameterized (by A) concretization of X and performing abstract fixpoint checking parameterized with this new element increases precision both in the abstract and in the concrete. Formally:
Let $A, B \in D$, $X = \mathrm{lfp}(\alpha\circ\alpha_A\circ f\circ\gamma_A\circ\gamma)$, $A \subseteq B \subseteq \gamma_A\circ\gamma(X)$ and $Y = \mathrm{lfp}(\alpha\circ\alpha_B\circ f\circ\gamma_B\circ\gamma)$. Then

1. $Y \sqsubseteq X$.
2. For $U = \gamma_A\circ\gamma(X)$ and $V = \gamma_B\circ\gamma(Y)$ holds $V \subseteq U$.

Proof. 1. From $B \subseteq A \cup \gamma(X)$ follows $B \cup \gamma(X) \subseteq A \cup \gamma(X)$. From $A \subseteq B$ follows $B^c \subseteq A^c$. We have $\alpha \circ \alpha_B \circ f \circ \gamma_B \circ \gamma(X) = \alpha((f(B \cup \gamma(X))) \cap B^c) \sqsubseteq$ [monotonicity of f and α] $\alpha((f(A \cup \gamma(X))) \cap A^c) = \alpha \circ \alpha_A \circ f \circ \gamma_A \circ \gamma(X) = X$. So X is a postfixpoint of $\alpha \circ \alpha_B \circ f \circ \gamma_B \circ \gamma$, while Y is its least fixpoint, which is by Tarski's fixpoint theorem less than or equal to any of its postfixpoints. Thus $Y \sqsubseteq X$.

2. From monotonicity of the concretization map follows $\gamma(Y) \subseteq \gamma(X)$ and thus $B \subseteq A \cup \gamma(X)$ implies $V = B \cup \gamma(Y) \subseteq A \cup \gamma(X) = U$.

□

For program analysis, this means that we can try to increase precision by choosing the new exception set between the current exception set and the program invariant obtained with the current exception set. This could be helpful, for instance, if a larger exception set can be represented more efficiently. If both A and B are contained in the least invariant, then the precision of abstract fixpoint checking is increased for sure:

Corollary 2.1.6. Let A, B be any two comparable elements below the least concrete fixpoint. Then parameterizing the abstract fixpoint computation by the greater element leads to a more precise abstract fixpoint than parameterizing it by the smaller element. Formally:
Let $A \subseteq B \subseteq \mathrm{lfp}(f)$, $X = \mathrm{lfp}(\alpha \circ \alpha_A \circ f \circ \gamma_A \circ \gamma)$, $Y = \mathrm{lfp}(\alpha \circ \alpha_B \circ f \circ \gamma_B \circ \gamma)$, $U = \gamma_A \circ \gamma(X)$, and $V = \gamma_B \circ \gamma(Y)$. Then $Y \sqsubseteq X$ and $V \subseteq U$.

Proof. The standard result of the abstract interpretation framework (cf. [1], Prop. 13) applied to $(\alpha \circ \alpha_A, \gamma_A \circ \gamma)$ implies $\mathrm{lfp}(f) \subseteq \gamma_A \circ \gamma(X)$. From $B \subseteq \mathrm{lfp}(f)$ follows $B \subseteq \gamma_A \circ \gamma(X)$. The result follows from Prop. 2.1.5. □

For program analysis this means that as long as the exception set grows, but remains below the best invariant, the precision of the analysis grows too. More precision can also be obtained from the following

Proposition 2.1.7. Assume some postfixpoint of f is a parameterized concretization of an abstract element. Then computation of the abstract fixpoint parameterized by the exception element gives an element whose parameterized concretization is less than or equal to that postfixpoint of f.

Formally: Let $E \in D$, $Y \in D^\#$, $R = \gamma_E \circ \gamma(Y)$, $f(R) \subseteq R$. Then $\gamma_E \circ \gamma \circ \mathrm{lfp}(\alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma) \subseteq R$.

Proof. Remember that for any Galois connection $(\bar{\alpha}, \bar{\gamma})$ between D and $D^\#$ holds that $\bar{\alpha} \circ \bar{\gamma}$ is reductive. Moreover both abstraction and concretization maps are monotonic. Then $\alpha_E(R) = R \cap E^c = (E \cup \gamma(Y)) \cap E^c = (E \cap E^c) \cup (\gamma(Y) \cap E^c) = \emptyset \cup (\gamma(Y) \cap E^c) \subseteq \gamma(Y)$. So $\gamma_E \circ \gamma \circ \alpha \circ \alpha_E(R) \subseteq \gamma_E \circ \gamma \circ \alpha \circ \gamma(Y) \subseteq$ [$\alpha \circ \gamma$ reductive] $\gamma_E \circ \gamma(Y) = R$. Then $\alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma(\alpha \circ \alpha_E(R)) = \alpha \circ \alpha_E \circ f(\gamma_E \circ \gamma \circ \alpha \circ \alpha_E(R)) \sqsubseteq \alpha \circ \alpha_E \circ f(R) \sqsubseteq$ [since $f(R) \subseteq R$] $\alpha \circ \alpha_E(R)$. So $\alpha \circ \alpha_E(R)$ is a postfixpoint of $\alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma$ and thus greater than or equal to its least fixpoint. So $\gamma_E \circ \gamma \circ \mathrm{lfp}(\alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma) \subseteq \gamma_E \circ \gamma \circ \alpha \circ \alpha_E(R) \subseteq R$. □

For program analysis, this means that if an inductive invariant of a program can be written as a parameterized concretization of an abstract element, then when the abstract fixpoint checking is parameterized by that parameter, the result is at least as good as the given invariant.

2.1.2 Polynomial-time parameterization

A description of what a parameterized abstract fixpoint is doesn't say anything about the implementability of the computation of such a fixpoint. Under certain conditions such a computation can be implemented in a fast way. We are going to give these conditions and say what "fast" means.

In order to be able to argue about run time, we measure a domain by its height and by the maximal size of the representation of its elements. We take the maximum of both values as the *measure* of the domain and formulate the run time as a function of this maximum.

Let's start with a

Hypothesis 2.1.8.
1. The computer model for time and space complexity is RAM with logarithmic cost measure and built-in addition (cf. [14]).
2. The map $f : D \to D$ is a join-morphism.
3. For a fixed E, the map $\alpha \circ \alpha_E \circ \gamma$ on $D^{\#+}$ is computable in polynomial time.
4. Abstract join of a finite number of abstract elements can be computed in polynomial time in the number and size of the elements.
5. The function f maps the concretization of every abstract element to a concrete join of concretizations of polynomially many abstract elements. Formally speaking, there exists a polynomial-time computable map

$$\bar{f} : D^{\#+} \to \mathfrak{P}(D^{\#+}), \quad y \mapsto Y,$$

such that if $|y|$ is the size of representation of y, then $|Y| \leq |y|^k$ for a constant $k \in \mathbb{N}$ independent on $|y|$ and such that

$$f \circ \gamma(y) = \bigsqcup \gamma(Y).$$

Condition 3 is satisfied, for instance, if all the abstract elements can be encoded as binary strings of finite length m; then that map can, as any operator on $\{0,1\}^m$, be encoded as a set of m decision trees. Alternative representations allowing poly-time checks are, for example, multi-terminal decision diagrams. The last condition 5 is the most restrictive one.

Proposition 2.1.9. For any domains D and $D^{\#}$, any f and any E satisfying the Hypothesis 2.1.8, one can precompute a data structure with only one call to f such that the best correct approximation $f^{\#}_E = \alpha \circ \alpha_E \circ f \circ \gamma_E \circ \gamma$ on $D^{\#+}$ is polynomial-time computable.

Proof. Let $y \in D^{\#+}$. Then

$$f^{\#}_E(y) = \alpha \circ \alpha_E \circ f \circ (E \sqcup \gamma(y)) = [\alpha \circ \alpha_E \circ f \text{ join-morphism}]$$
$$= \alpha \circ \alpha_E \circ f(E) \sqcup \alpha \circ \alpha_E \circ f \circ \gamma(y)$$

Since E is a constant independent on y, the expression $\alpha \circ \alpha_E \circ f(E) =: E'$ can be computed once with one call to f; we view E' as a part of representation of E. The join operation needs polynomial time, so is suffices to show that the map $\alpha \circ \alpha_E \circ f \circ \gamma$ is polynomial-time computable. The hypothesis implies that

$$\alpha \circ \alpha_E \circ f \circ \gamma(y) = \alpha \circ \alpha_E(\bigsqcup \gamma(\bar{f}(y))) =$$
$$[\alpha \circ \alpha_E \text{ join-morphism}] \bigsqcup \{\alpha \circ \alpha_E \circ \gamma(y') \mid y' \in \bar{f}(y)\}.$$

The set $\bar{f}(y)$ has polynomial many elements, the abstract join $\bigsqcup$ is polynomial-time computable, and each $\alpha \circ \alpha_E \circ \gamma(y')$ is polynomial-time computable ($y' \in \bar{f}(y)$). Thus $\alpha \circ \alpha_E \circ f \circ \gamma$ is polynomial-time computable. □

Now we turn to program analysis.

Theorem 2.1.10. Assume a finite-state program with a set of initial states init and successor map post and E such that the Hypothesis 2.1.8 is satisfied for post. Then it is possible to precompute a data structure with only one call to post and one call of $\alpha \circ \alpha_E$ so that abstract fixpoint computation takes polynomial time in the measure of the abstract domain. Formally:
Under the mentioned conditions computing lfp $(\lambda Y.\ \alpha \circ \alpha_E(\text{init} \cup \text{post} \circ \gamma_E \circ \gamma(Y)))$ takes polynomial time in the measure of the abstract domain.

Proof. Take the abstract fixpoint iteration sequence $X^0 = \bot$ and $X^{i+1} = \alpha \circ \alpha_E(\text{init} \cup \text{post} \circ \gamma_E \circ \gamma(X^i))$. This sequence is monotone and stabilizes in a finite number of steps. The abstraction maps α and α_E are join-morphisms, so $X^{i+1} = \alpha \circ \alpha_E(\text{init}) \sqcup \alpha \circ \alpha_E \circ \text{post} \circ \gamma_E \circ \gamma(X^i)$. The first term is constant, the second can be computed in polynomial time as just shown, the join operation is polynomial. So the next term of the sequence $(X^i)_i$ is computable in polynomial time from the previous one. Since the maximal chain length is polynomial, the number of iterations till the sequence stabilizes is also polynomial. □

2.2 Parameterized Cartesian abstraction

Now we parameterize the multithreaded Cartesian Galois connection. We show how to compute the parameterized multithreaded Cartesian abstract post, i.e.

$$\text{post}^{\#}_{\text{mc},E} = \alpha_{\text{mc}} \circ \alpha_E \circ \text{post} \circ \gamma_E \circ \gamma_{\text{mc}}\,,$$

examine the complexity of the corresponding abstract fixpoint checking, and look at examples.

The algorithm for computing the parameterized abstract post depends on the representation of the argument and of E. First we reduce the computation of $\text{post}^{\#}_{\text{mc},E}$ to computations of conceptually simpler "small" expressions of a particular form. Then we show the complexity of the difficult problem assuming an oracle for "small" expressions. At last we show how to deal with the "small" expressions. We assume that programs are finite-state.

We represent sets of n-tuples as unions of products. A set of states S is thus represented as a list of tuples $(S_i^{(g,j)})_{g \in \text{Glob}, j \in \mathbb{N}_m, i \in \mathbb{N}_n}$ such that

$$S = \bigcup_{g \in \text{Glob}} \left(\{g\} \times \underbrace{\bigcup_{j=1}^{m} \prod_{i=1}^{n} S_i^{(g,j)}}_{S\downarrow g} \right).$$

A set of n-tuples corresponding to a set of states S and a shared state g is written as

$$S\downarrow g \;=\; \{l \in \text{Loc}^n \mid (g,l) \in S\}\,.$$

2.2.1 Polynomial reduction to inclusion tests

We show how to reduce the computation of the abstract program semantics to polynomially many computations of simpler expressions.

The iterative computation of $\mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y))$ needs at most as many evaluations of the parameterized abstract post operator as the height of the abstract lattice, which is polynomial. It suffices to show how to compute $\mathrm{post}^{\#}_{\mathrm{mc},E}$.

Proposition 2.2.1. Computation of $\mathrm{post}^{\#}_{\mathrm{mc},E}(Y)$ is polynomially reducible to computations of $\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z) \setminus F)$ for some $Z \in (\mathfrak{P}(\mathrm{Loc}))^n$ and some $F \subseteq \mathrm{Loc}^n$ such that F is represented as a union of Cartesian products in space not exceeding the size of representation of E.

Idea. Construct an intermediate problem: assume for simplicity that in Y only one shared state occurs. The reductions follow from the definition of the successor map and the fact that the successor map and the abstraction map are join-morphisms.

Proof. Let $E \in D$ be represented by products $E^{(g,j)} \subseteq \mathrm{Loc}^n$ ($g \in \mathrm{Glob}$, $1 \le j \le m(g)$) and Y by sets $Y_i^{(g)} \subseteq \mathrm{Loc}$ ($g \in \mathrm{Glob}$, $i \in \mathbb{N}_n$) as

$$E = \bigcup_{g \in \mathrm{Glob}} \left(\{g\} \times \underbrace{\bigcup_{j=1}^{m(g)} E^{(g,j)}}_{E\downarrow g} \right), \qquad Y = \left(\bigcup_{g \in \mathrm{Glob}} \{g\} \times Y_i^{(g)} \right)_{i=1}^{n}.$$

Since post and $\alpha_{\mathrm{mc}} \circ \alpha_E$ are join-morphisms,

$$\alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post} \circ \gamma_E \circ \gamma_{\mathrm{mc}}(Y) =$$

$$\alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(E) \sqcup \alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\gamma_{\mathrm{mc}}(Y)) = \quad [\text{where } Y^{(g)} = \prod_{i=1}^{n} Y_i^{(g)}]$$

$$\bigsqcup_{g \in \mathrm{Glob}} \left(\alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\{g\} \times E\downarrow g) \sqcup \alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\{g\} \times Y^{(g)}) \right) =$$

$$\bigsqcup_{g \in \mathrm{Glob}} \left(\bigsqcup_{j=1}^{m(g)} \alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\{g\} \times E^{(g,j)}) \sqcup \alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\{g\} \times Y^{(g)}) \right).$$

It suffices to polynomially often evaluate expressions of the form

$$\alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\{g\} \times X)$$

for products $X = \prod_{i=1}^{n} X_i \subseteq \mathrm{Loc}^n$. For computing $\mathrm{post}(\{g\} \times X)$ it suffices to consider only transitions of the form $((g,l),(g',l'))$ of some thread i for some $g' \in \mathrm{Glob}$ and $l, l' \in \mathrm{Loc}$. Let $\mathrm{post}_{i,((g,l),(g',l'))}$ be the successor map for such a transition only. Then

$$\alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}(\{g\} \times X) = \bigsqcup_{\substack{1 \le i \le n \\ ((g,l),(g',l')) \in \to_i}} \alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}_{i,((g,l),(g',l'))}(\{g\} \times X).$$

If $l \in X_i$, then $\alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post}_{i,((g,l),(g',l'))}(\{g\} \times X)$ is equal to $\alpha_{\mathrm{mc}} \circ \alpha_E(\{g'\} \times X_1 \times \ldots \times X_{i-1} \times \{l'\} \times X_{i+1} \times \ldots \times X_n)$, otherwise it is equal to $\alpha_{\mathrm{mc}} \circ \alpha_E(\emptyset) = \bot$.

We have to evaluate polynomially many expressions of the form

$$\alpha_{\mathrm{mc}}((\{g'\} \times \gamma_{\mathrm{c}}(Z)) \setminus E)\,,$$

(for some product $\gamma_{\mathrm{c}}(Z)$) which is equal to $\alpha_{\mathrm{mc}}((\{g'\} \times \gamma_{\mathrm{c}}(Z)) \setminus (\{g'\} \times E{\downarrow}g')) = \alpha_{\mathrm{mc}}(\{g'\} \times (\gamma_{\mathrm{c}}(Z) \setminus E{\downarrow}g'))$. By definition of α_{mc}, it suffices to evaluate

$$\alpha_{\mathrm{c}}\left(\gamma_{\mathrm{c}}(Z) \setminus E{\downarrow}g'\right)\,,$$

prepending g' to each local state in each component of the result. □

Notice that F depends on E only, so it can be precomputed for a fixed E.

Proposition 2.2.2. Computing $\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z) \setminus F)$ for $Z \in (\mathfrak{P}(\mathrm{Loc}))^n$ and $F \subseteq \mathrm{Loc}^n$, represented as a union of Cartesian products, is polynomially reducible to evaluation of expressions of the form $\gamma_{\mathrm{c}}(A) \stackrel{?}{\subseteq} B$ for $A \in (\mathfrak{P}(\mathrm{Loc}))^{n-1}$ and $B \subseteq \mathrm{Loc}^{n-1}$, represented as a union of Cartesian products.

Idea. Compute each component of the resulting tuple of sets separately by checking for each local state whether it should be in that component.

Proof. Define for each $i \in \mathbb{N}_n$ and each $r \in \mathrm{Loc}$ the set $F_{i,r} = \{f \in \mathrm{Loc}^{n-1} \mid f \cup \{(i,r)\} \in F\}$.

Claim: Whenever $r \in \mathrm{Loc}$ and $i \in \mathbb{N}_n$, we have

$$r \in (\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z) \setminus F))_i \quad \Leftrightarrow \quad r \in Z_i \text{ and } \prod_{j \in \mathbb{N}_n \setminus \{i\}} Z_j \not\subseteq F_{i,r}\,.$$

To prove "⇒", let $r \in (\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z) \setminus F))_i = \pi_i(\gamma_{\mathrm{c}}(Z) \setminus F)$. So there is an n-tuple $z \in F^c \cap \prod_{i=1}^n Z_i$ with $z_i = r$, thus $r \in Z_i$. Moreover $z \notin F$ and $z_j \in Z_j$ $(j \in \mathbb{N}_n)$. So the $(n-1)$-tuple $z \setminus \{(i,r)\} \in \prod_{j \in \mathbb{N}_n \setminus \{i\}} Z_j$, but $z \setminus \{(i,r)\} \notin F_{i,r}$.

To prove "⇐", let $r \in Z_i$ and let z be an $(n-1)$-tuple with $z \in \prod_{j \in \mathbb{N}_n \setminus \{i\}} Z_j$ and $z \notin F_{i,r}$. Then $z \cup \{(i,r)\} \notin F$, but $z \cup \{(i,r)\} \in \prod_{j=1}^n Z_j = \gamma_{\mathrm{c}}(Z)$. Thus $z \cup \{(i,r)\} \in \gamma_{\mathrm{c}}(Z) \setminus F$, hence $r \in \pi_i(\gamma_{\mathrm{c}}(Z) \setminus F) = (\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z) \setminus F))_i$.

The claim is proven. Now notice that if F is represented by sets $F_i^{(j)} \subseteq \mathrm{Loc}$ $(j \in \mathbb{N}_m,\ i \in \mathbb{N}_n)$ as $F = \bigcup_{j=1}^m \prod_{i=1}^n F_i^{(j)}$, then for each $i \in \mathbb{N}_n$ and $r \in \mathrm{Loc}$ we have $F_{i,r} = \bigcup_{\substack{j \in \mathbb{N}_m \\ r \in F_i^{(j)}}} \prod_{\substack{k \in \mathbb{N}_n \\ k \neq i}} F_k^{(j)}$. So the expressions are of the required form $\gamma_{\mathrm{c}}(A) \subseteq B$ where B is a union of products. To compute $\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z) \setminus F)$, it suffices to check for each $i \in \mathbb{N}_n$ and each $r \in \mathrm{Loc}$, whether $r \in (\alpha_{\mathrm{c}}(\gamma_{\mathrm{c}}(Z \setminus F)))_i$. So the number of expressions is polynomial. □

Corollary 2.2.3. Computing the abstract fixpoint semantics under the parameterized Cartesian Galois connection is polynomial in the number of inclusion tests. Formally, computing

$$\mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y)) \tag{2.1}$$

is reducible to evaluation of polynomially many expressions of the form $\gamma_{\mathrm{c}}(A) \stackrel{?}{\subseteq} B$ where B is represented as a union of products.

The exception set before and after post need not be the same. We will later need the following

Remark 2.2.4. Following the lines of the proof of Prop. 2.2.1, and using Prop. 2.2.2 one can show that computing the operator

$$\alpha_{\mathrm{mc}} \circ \alpha_{E'} \circ \mathrm{post} \circ \gamma_E \circ \gamma_{\mathrm{mc}}$$

is also polynomially reducible to evaluation of expressions of the form $\gamma_{\mathrm{c}}(A) \stackrel{?}{\subseteq} B$, such that for each shared part, the sets of corresponding local parts of E and E' are represented as a union of products, and B is represented as a union of products.

Expressions $\gamma_{\mathrm{c}}(A) \stackrel{?}{\subseteq} B$ can be evaluated efficiently in practice. However, before considering an algorithm for this subproblem, we'll study the complexity of the fixpoint computation assuming an oracle for evaluating the expressions.

2.2.2 Complexity

We are going to determine the complexity of (2.1) and of related problems.

Proposition 2.2.5. Determining whether $A \subseteq B$, where A is a product and B is a union of products of the same dimension, is a coNP-complete problem.

Formally, the language

$$\mathrm{INC} := \left\{ \left\langle (A_i)_{i\in\mathbb{N}_n}, (B_j^{(j)})_{\substack{i\in\mathbb{N}_n \\ j\in\mathbb{N}_m}} \right\rangle \;\middle|\; \prod_{i=1}^{n} A_i \subseteq \bigcup_{j=1}^{m} \prod_{i=1}^{n} B_i^{(j)} \right\}$$

is coNP-complete (we write INC for inclusion problem).

The angle brackets $\langle \ldots \rangle$ denote an encoding into a fixed alphabet.

Idea. Use guessing for membership and boolean tautology for hardness.

Proof. On one hand, to prove $\prod_{i=1}^{n} A_i \nsubseteq \bigcup_{j=1}^{m} \prod_{i=1}^{n} B_i^{(j)}$, it suffices to guess a tuple that it is in $\prod_{i=1}^{n} A_i$ but not in $\prod_{i=1}^{n} B_i^{(j)}$ for any $1 \le j \le m$. So INC is in coNP.

On the other hand, one might encode the tautology problem for disjunctive normal forms as inclusion. For that, take a disjunctive normal form $\bigvee_{j=1}^{m} T^{(j)}$ where each $T^{(j)}$ $(1 \le j \le m)$ is of the form $\bigwedge_{i\in I_j} y_i^{(j)}$ where each $y_i^{(j)}$ is in $\{x_i, \neg x_i\}$ $(i \in I_j \subseteq \mathbb{N}_n)$. Transform each term $T^{(j)}$ into a product $B^{(j)}$, setting

$$B_i^{(j)} = \begin{cases} \{0\} & \text{if } \neg x_i \text{ in } T^{(j)}\,, \\ \{1\} & \text{if } x_i \text{ in } T^{(j)}\,, \\ \{0,1\} & \text{otherwise}\,. \end{cases}$$

Set $A = \{0,1\}^n$. The disjunctive normal form is a tautology if and only if $A \subseteq \bigcup_{j=1}^{m} \prod_{i=1}^{n} B_i^{(j)}$. So INC is coNP-hard. □

Viewing the computation as a function problem, we obtain directly

Corollary 2.2.6. Computing the abstract fixpoint semantics under multithreaded Cartesian Galois connection is polynomial given an NP-oracle. Formally:
The map that, for a program $(\mathrm{Glob}, \mathrm{Loc}, (\to_i)_{i=1}^n, \mathrm{init})$ and a set of states E, returns

$$\mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y))\,,$$

is in $\mathrm{FP}^{\mathrm{NP}}$.

Corollary 2.2.7. Given two sets S and T of products of the same dimension, checking $\bigcup S \subseteq \bigcup T$ is coNP-complete.

Proof. In order to check non-inclusion, it suffices to guess a tuple a and a product $A \in S$ such that $a \in A$ but $a \notin B$ for any B in T. CoNP-hardness follows by reduction from INC, taking a singleton S. □

We can also look at the complexity of the corresponding decision problem. We define the *parameterized abstract fixpoint checking* (of multithreaded programs with parameterized multithreaded Cartesian abstraction and unions of products as data structure) as the language

$$\mathrm{PAFC} = \Bigg\{ \langle P,\, E, \mathrm{safe}\rangle \mid P = (\mathrm{Glob}, \mathrm{Loc}, (\to_i)_{i=1}^n, \mathrm{init}) \text{ is a program with successor map post and } E, \mathrm{safe} \text{ are sets of its states and } \gamma_E \circ \gamma_{\mathrm{mc}} \circ \mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y)) \;\subseteq\; \mathrm{safe} \Bigg\}.$$

By Corl. 2.2.6, $\mathrm{PAFC} \in \Delta_2^p = \mathrm{P}^{\mathrm{NP}}$. However a much stronger bound holds:

Theorem 2.2.8. Parameterized abstract fixpoint checking is coNP-complete.

Idea. To show coNP-hardness, reduce the inclusion problem to PAFC. To show containment in coNP, guess the least abstract fixpoint iteration sequence in a manner which is sound for non-inclusion.
Proof. To prove coNP-hardness, reduce from INC. Take an instance $\langle A, B\rangle$ of INC where A defines a product and B a union of products of the same dimension. Set $\mathrm{init} = \to_i = \emptyset$ for all $i \in \mathbb{N}_n$ (effectively making the least fixed point empty), $E = A$ and $\mathrm{safe} = B$.

Membership in coNP requires showing that the complement of the language is in NP. We give a nondeterministic algorithm that takes an input and tests in nondeterministic polynomial time that the input does not belong to PAFC.

Checking syntax can be done without using guesses. Now assume that the input has a correct syntax $\langle P, E, \mathrm{safe}\rangle$ such that $P = (\mathrm{Glob}, \mathrm{Loc}, (\to_i)_{i=1}^n, \mathrm{init})$ is a program with successor map post and E, safe are sets of its states. Now we have to test using nondeterministic guesses whether

$$\gamma_E \circ \gamma_{\mathrm{mc}} \circ \mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y)) \;\not\subseteq\; \mathrm{safe}\,.$$

First guess one bit to determine whether $E \not\subseteq \mathrm{safe}$ or $\gamma_{\mathrm{mc}} \circ \mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y)) \not\subseteq \mathrm{safe}$. In the first case guess a state in E and test that it doesn't belong to any product in the representation of safe. Otherwise guess a chain

$$\bot = Y^{(0)} \sqsubset Y^{(1)} \sqsubset \ldots \sqsubset Y^{(m)}$$

of length $m+1 \leq \mathfrak{h}(D^\#, \sqsubseteq) = |\text{Glob}||\text{Loc}|+1$ such that $Y^{(j)} \in D^\#$ for all $j \leq m$ and test that $\alpha_{\text{mc}} \circ \alpha_E(\text{init}) \sqcup \text{post}^\#_{\text{mc},E}(Y^{(j)}) \sqsupseteq Y^{(j+1)}$ for all $j < m$ and $\gamma_E \circ \gamma_{\text{mc}}(Y^{(m)}) \not\subseteq$ safe. To test $\gamma_E \circ \gamma_{\text{mc}}(Y^{(j)}) \not\subseteq$ safe, guess a state in $E \cup \gamma_{\text{mc}}(Y^{(m)})$ and check that it doesn't belong to safe.

To test $\alpha_{\text{mc}} \circ \alpha_E(\text{init}) \sqcup \text{post}^\#_{\text{mc},E}(Y^{(j)}) \sqsupseteq Y^{(j+1)}$ for a fixed $j < m$, guess two abstract elements $A, B \in D^\#$ and test that $\alpha_{\text{mc}} \circ \alpha_E(\text{init}) \sqsupseteq A$ and $\text{post}^\#_{\text{mc},E}(Y^{(j)}) \sqsupseteq B$ and $A \sqcup B \sqsupseteq Y^{(j+1)}$.

To test $\alpha_{\text{mc}} \circ \alpha_E(\text{init}) \sqsupseteq A$, it suffices to do it for each shared state $g \in \text{Glob}$, i.e. to test whether $\alpha_{\text{c}} \circ \alpha_{E\downarrow g}(\text{init}\downarrow g) \sqsupseteq A\downarrow g$. Fix some g. Notice that $\text{init}\downarrow g$ is represented as $\bigcup_{1\leq k\leq q(g)} \text{init}^{(g,k)}$ for products $\text{init}^{(g,k)}$. So it suffices to guess for each $1 \leq k \leq q(g)$ products $A^{(g,k)}$ and test that $\alpha_{\text{c}} \circ \alpha_{E\downarrow g}(\text{init}^{(g,k)}) \sqsupseteq A^{(g,k)}$ for all k and that $\bigsqcup_{k=1}^{q(g)} A^{(g,k)} \sqsupseteq A\downarrow g$.

Later we will show how to execute such a test, i.e. a test of the form

$$\alpha_{\text{c}} \circ \alpha_F(G) \sqsupseteq H \tag{2.2}$$

where H and G are products and F is a sets of tuples of the same dimension.

To test whether $\text{post}^\#_{\text{mc},E}(Y^{(j)}) \sqsupseteq Y^{(j+1)}$, it suffices to guess two abstract elements I, J and test that $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post}(E) \sqsupseteq I$ and $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post} \circ \gamma_{\text{mc}}(Y^{(j)}) \sqsupseteq J$ and that $I \sqcup J \sqsupseteq Y^{(j+1)}$.

To test $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post}(E) \sqsupseteq I$, we take the representation of the exception set $E = \bigcup_{g\in\text{Glob}}\{g\} \times \bigcup_{k=1}^{p(g)} E^{(g,k)}$. It suffices to guess abstract elements $K^{(g,1)}, \ldots, K^{(g,p(g))}$ for each $g \in \text{Glob}$ and test for all g and all $1 \leq k \leq p(g)$ that $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post}(\{g\} \times E^{(g,k)}) \sqsupseteq K^{(g,k)}$ and that $\bigsqcup_{g\in\text{Glob},1\leq k\leq p(g)} K^{(g,k)} \sqsupseteq I$.

To test $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post} \circ \gamma_{\text{mc}}(Y^{(j)}) \sqsupseteq J$, it suffices to guess abstract elements $L^{(g)}$ for each $g \in \text{Glob}$ and then to test that $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post}(\{g\} \times Y^{(j)}\downarrow g) \sqsupseteq L^{(g)}$ for all g and that $\bigsqcup_{g\in\text{Glob}} L^{(g)} \sqsupseteq J$.

Now to test whether $\alpha_{\text{mc}} \circ \alpha_E \circ \text{post}(\{g\} \times M) \sqsupseteq N$ for a product M and an abstract element N, it suffices to guess for each transition $((g,l),(g',l')) \in \to_i$ of each thread i an abstract element $O^{(i,g,l,g',l')}$ and test that for the successor operator with respect to that transition only we have

$$\alpha_{\text{mc}} \circ \alpha_E \circ \text{post}_{i,g,l,g',l'}(\{g\} \times M) \sqsupseteq O^{(i,g,l,g',l')}\,, \tag{2.3}$$

and that $\bigsqcup_{i\in\mathbb{N}_n,\,(g,l)\to_i(g',l')} O^{(i,g,l,g',l')} \sqsupseteq N$. If M is empty, the left hand side of (2.3) is empty, so testing the $\sqsupseteq$ relation is equivalent to testing $\bot = N$. Otherwise write $M = \prod_{k=1}^n M_k$. If $l \notin M_i$, then the left hand side is also empty. If $l \in M_i$, it suffices to check that $\alpha_{\text{mc}} \circ \alpha_E(\{g'\} \times M_1 \times \ldots \times M_{i-1} \times \{l'\} \times M_{i+1} \times \ldots \times M_n) \sqsupseteq O^{(i,g,l,g',l')}$.

To check that $\alpha_{\text{mc}} \circ \alpha_E(\{g'\} \times R) \sqsupseteq Q$ for a product R and an abstract element Q, it suffices to check $\alpha_{\text{c}} \circ \alpha_{E\downarrow g'}(R) \sqsupseteq Q\downarrow g'$.

Now it suffices to show how to test forms like (2.2), i.e.

$$\alpha_{\text{c}}(G \setminus F) \sqsupseteq H\,,$$

where $G = \prod_{k=1}^n G_k$ and $H = \prod_{k=1}^n H_k$ are products and $F = \bigcup_{v=1}^s \prod_{k=1}^n F_k^{(v)}$ a union of products. It suffices to test for each $i \in \mathbb{N}_n$ and $r \in H_i$, whether $r \in \pi_i(G \setminus F)$. Fix such i and r. By the claim in the proof of Prop. 2.2.2, it suffices to test whether $r \in G_i$ and whether $\prod_{k\neq i} G_k \not\subseteq F_{i,r}$ where $F_{i,r} =$

$\bigcup_{\substack{1\le v\le s \\ r\in F_i^{(v)}}} \prod_{\substack{1\le k\le n \\ k\neq i}} F_k^{(v)}$.
Testing whether a product on the left hand side is not contained in a union of products on the right hand side can be done by guessing a tuple in the left hand side and checking that it is not in any product on the right hand side. □

So we managed to reduce complexity of safety checking (PSPACE-complete by a reduction from nonemptiness of regular language intersection, cf. [9]) to coNP by introducing a parameter to multithreaded Cartesian abstract fixpoint checking.

2.2.3 Complexity for maximized representation

A set of n-tuples may be represented in different ways as a union of products. We show one way which leads to a polynomial complexity, improving the coNP result. The runtime of computation of the parameterized abstract fixpoint and of the parameterized abstract fixpoint checking gets polynomial. We are going to define the representation and sketch the main proof ideas.

Definition 2.2.9. A set of Cartesian products M is called *maximized* if each product Y that is a subset of $\bigcup M$ is a subset of an element of M, formally

$$\forall Y: \quad Y \text{ is a product and } Y \subseteq \bigcup M \;\Rightarrow\; \exists Y' \in M : Y \subseteq Y' .$$

A set of products M *represents* a set of tuples X if $\bigcup M = X$.

One can show that each set of tuples can be represented by a maximized set of products. Moreover, one can require non-redundant representation as maximized antichains, i.e. no two products in the set are comparable. Each set of tuples has a unique representation as a maximized antichain of products. The products in this antichain are exactly the maximal products contained the union of the antichain.

Assume that E and safe are represented in this way. Namely, assume that for each $g \in \mathrm{Glob}$ the set $E{\downarrow}g$ as well as $\mathrm{safe}{\downarrow}g$ is represented by a maximized antichain of products.

Transformations described in Subsection 2.2.1 work independently of the fact that the set of products E is a maximized antichain. Moreover, if E is represented as a union over a maximized set, then one can show that in all the used expressions $\gamma_c(A) \subseteq B$ the set B is represented as a union over a maximized set. Now testing inclusion in a union over a maximized set M of products is very easy:

$$M \text{ maximized and } \gamma_c(A) \subseteq \bigcup M \quad\Rightarrow\quad \gamma_c(A) \subseteq B \text{ for some } B \in M .$$

Since the computation of the parameterized abstract fixpoint reduces to polynomially many evaluations of expressions which can be solved for this representation in polynomial time, the whole computation of the parameterized abstract fixpoint

$$\mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y))$$

is polynomial. Checking safety requires checking that E and the above fixpoint are subsets of safe. This can be done by checking for each $g \in \mathrm{Glob}$ that

$$E{\downarrow}g \subseteq \mathrm{safe}{\downarrow}g \;\text{ and }\; \gamma_{\mathrm{mc}}(\mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y))){\downarrow}g \subseteq \mathrm{safe}{\downarrow}g .$$

Both last checks are polynomial since safe$\downarrow g$ is represented as a maximized set. We obtain

Theorem 2.2.10. Computation of the parameterized abstract fixpoint and parameterized abstract fixpoint checking are polynomial-time if the exception set and the safety property are represented in such a way that for each shared state the corresponding set of local states is represented by a maximized set.

Formally, if for each $g \in \mathrm{Glob}$ the sets $E\downarrow g$ and safe$\downarrow g$ are represented as maximized sets of Cartesian products, then

$$\Big(\langle P, E, \mathrm{safe}\rangle \mapsto \gamma_E \circ \gamma_{\mathrm{mc}} \circ \mathrm{lfp}(\lambda Y.\, \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \mathrm{post}^{\#}_{\mathrm{mc},E}(Y))\Big) \quad \in \quad \mathrm{FP}$$
$$\text{and} \quad \mathrm{PAFC} \quad \in \quad \mathrm{P}\,.$$

After having seen another representation we ask whether there is an algorithm that needs at most polynomial space (in the program size and $|E|$) for the exception set and at most polynomial time (in the program size). We believe that the core of the question lies in the following

Problem 2.2.11. Is there a unary polynomial p over integers so that for all natural numbers n, d and for each subset $B \subseteq (\mathbb{N}_d)^n$ there is an algorithm (a deterministic Turing machine or a deterministic RAM program with a data structure, with logarithmic cost measure and built-in logarithmic cost addition) of size $\leq p(d)^{p(n)}$ that, in time $\leq p(dn)$, given an n-tuple $(A_i)_{i=1}^n$ of subsets of $\mathbb{N}_d$ as an input, answers the question whether $\prod_{i=1}^n A_i \subseteq B$?

2.2.4 Generating the maximized representation

Now we show how to generate a maximized representation of a set of n-tuples. It turns out that, given a finite set of products of the same dimension, one can generate a maximized set of products by repeatedly applying the so-called *Cartesian product resolution* rule, which we will describe.

For any set M of n-dimensional products, any $X = \prod_{i=1}^n X_i$ in M, any $Y = \prod_{i=1}^n Y_i$ in M and any $j \in \mathbb{N}_n$, let $R(X, Y, j) =$

$$\begin{cases} & (X_1 \cup Y_1) & \times & \prod_{1<i\leq n}(X_i \cap Y_i)\,, & \text{if } j = 1\,, \\ \prod_{1\leq i<j}(X_i \cap Y_i) & \times & (X_j \cup Y_j) & \times & \prod_{j<i\leq n}(X_i \cap Y_i)\,, & \text{if } 1 < j < n\,, \\ \prod_{1\leq i<n}(X_i \cap Y_i) & \times & (X_n \cup Y_n)\,, & & & \text{if } j = n\,. \end{cases}$$

We call $R(X, Y, j)$ the j^{th} *resolvent* of X and Y.

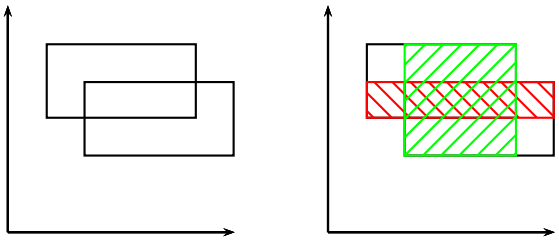
The following procedure `resolve` adds, if possible, a resolvent to M that is not a subset of a product of M. If M is empty, so is `resolve`(M).

Input: M: finite set of products of the same dimension n
Output: finite set of n-dimensional products
if $\exists\, X \in M, Y \in M, j \in \mathbb{N}_n : \forall\, C \in M :\ R(X, Y, j) \not\subseteq C$ **then**
  choose any such X, Y, j;
  return $M \cup \{R(X, Y, j)\}$;
else
  return M;

Algorithm 2: `resolve`

To generate a maximized representation of M we saturate M by applying `resolve` repeatedly, i.e. by computing $\texttt{resolve}^i(M)$ such that $\texttt{resolve}^i(M) = \texttt{resolve}^{i+1}(M)$. One can show that for a finite set M such an i exists, i.e. one needs only finitely many applications. Moreover, one can show that the saturated set is maximized. We give reasons for those facts without going into proof details. The reason for termination is that components of the products can be separated into only finitely many regions by taking unions and intersections. These regions can be combined into products in only finitely many ways. To show that the saturated set is a maximized one, one uses induction over the dimension number, for each dimension and any product A contained in the union constructing a product in the saturated set which contains A.

In the example on Fig. 2.1a, the set of two rectangles is taken and saturated by the resolve procedure, due to which two new products appear on Fig. 2.1b. One created product is the first resolvent, i.e. union is taken over the first component and intersection over the second. The other created product is the second resolvent, in which union is taken over the second component and intersection over the first.



(a) M (b) saturated M

Figure 2.1: `resolve` applied twice till saturation

As an optimization, one might remove subsumed elements at any time point in the computation, i.e. those elements that are included in smaller ones. The following procedure `remove_subsumed` removes, if possible, a product from M that is a subset of a different product of M.

Input: M: finite set of products of the same dimension n
Output: finite set of n-dimensional products
if $\exists X \in M : \exists Y \in M : X \subsetneq Y$ **then**
  choose any such X;
  return $M \setminus \{X\}$;
else
  return M;

Algorithm 3: `remove_subsumed`

Applying `remove_subsumed` repeatedly till fixpoint results in an antichain. If a set of products M satisfies $\texttt{remove\_subsumed}(M) = \texttt{resolve}(M) = M$, then M is a maximized antichain.

The author's optimized implementation [10] generates the maximized representation on demand and stores the generated maximized antichains from one

inclusion test to another, saving computation time.

2.2.5 Examples

Peterson's mutual exclusion

We illustrate the refined thread-modular algorithm for precise thread-modular verification on Peterson's mutual exclusion algorithm shown in Fig. 2.2. We wish to verify that at most one thread is at location D.

$$\textbf{shared } x = y = turn = 0$$

$$P_1 :: \left[\begin{array}{ll} A\colon & x := 1; \\ B\colon & turn := 1; \\ C\colon & \textbf{while}(y \textbf{ and } turn); \\ & \textbf{critical} \\ D\colon & x := 0; \textbf{ goto } A; \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{ll} A\colon & y := 1; \\ B\colon & turn := 0; \\ C\colon & \textbf{while}(x \textbf{ and not } turn); \\ & \textbf{critical} \\ D\colon & y := 0; \textbf{ goto } A; \end{array}\right]$$

Figure 2.2: Peterson's mutual exclusion algorithm.

First let us compute an over-approximation of the reachable states by applying the Flanagan-Qadeer thread-modular verification algorithm without exception sets. The result is represented by the following union of Cartesian products:

$$\begin{array}{lllll} & \{000\} & \times\{A\} & & \times\{A\} \\ \cup & \{001\} & \times\{A\} & & \times\{A\} \\ \cup & \{010\} & \times\{A\} & & \times\{B,C,D\} \\ \cup & \{011\} & \times\{A\} & & \times\{B,C,D\} \\ \cup & \{100\} & \times\{B,C,D\} & & \times\{A\} \\ \cup & \{101\} & \times\{B,C,D\} & & \times\{A\} \\ \cup & \{110\} & \times\{B,C,D\} & & \times\{B,C,D\} \\ \cup & \{111\} & \times\{B,C,D\} & & \times\{B,C,D\}\,, \end{array}$$

where abc (e.g. 011) denotes the shared part $x = a \wedge y = b \wedge turn = c$ (e.g. $x = 0 \wedge y = 1 \wedge turn = 1$). This over-approximation is too coarse. It contains some states where both the first and the second thread are at their locations D, namely $(111, D, D)$ and $(110, D, D)$ (i.e. $x = y = turn = 1 \wedge pc_1 = pc_2 = D$ and $x = y = 1 \wedge turn = 0 \wedge pc_1 = pc_2 = D$).

Now we apply the refined algorithm instead. It iteratively computes an over-approximation of the reachable states, without losing the dependencies between the successors of the states contained in a given exception set. Let $E = \{(110, B, D), (110, C, C), (111, D, B)\}$ be the exception set.

We start the iteration with the initial state set $X_1 = \{(000, A, A)\}$. We compute the over-approximation of the set of states that are reachable from it in one step as follows. First, we take the smallest Cartesian product that contains the local parts of $(000, A, A)$. This is again

$$\{000\} \times \{A\} \times \{A\}\,.$$

Then we make a step which is specific to our algorithm. We extend this set by adding the elements of the exception set, which yields

$$X_2 = \{(000, A, A), (110, B, D), (110, C, C), (111, D, B)\}\,.$$

For this set, we compute the image under the one-step reachability under post, which is induced by the program, and add the initial element, which is in X_1. The resulting set is $\{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, C, C), (110, D, C), (111, C, D)\}$. Before over-approximating this set, we perform another step that is specific to our algorithm. We subtract the exception set from the result, which yields the set $\{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, D, C), (111, C, D)\}$. Then, for each shared part, we take the smallest Cartesian product that contains the local parts. This gives again the same set

$$\begin{array}{lllll} & \{000\} & \times\{A\} & \times\{A\} \\ \cup & \{010\} & \times\{A\} & \times\{B\} \\ \cup & \{011\} & \times\{A\} & \times\{B\} \\ \cup & \{100\} & \times\{B\} & \times\{A\} \\ \cup & \{110\} & \times\{D\} & \times\{C\} \\ \cup & \{111\} & \times\{C\} & \times\{D\}\,. \end{array}$$

At last, we restore the states which get excluded before over-approximations, obtaining

$$X_3 = \{(000, A, A), (010, A, B), (011, A, B), (100, B, A), (110, B, D), (110, C, C), (110, D, C), (111, C, D), (111, D, B)\}\,.$$

We continue the fixpoint computation by applying the standard steps interleaved with the specific steps. The standard steps are taking one-step-successors and adding the initial states. The specific steps are subtracting the exception set away, applying the over-approximation and adding the exception set back.

The fixpoint of the described procedure is

$$\begin{array}{lllll} X_5 = & \{000\} & \times\{A\} & \times\{A\} \\ \cup & \{001\} & \times\{A\} & \times\{A\} \\ \cup & \{010\} & \times\{A\} & \times\{B, C, D\} \\ \cup & \{011\} & \times\{A\} & \times\{B\} \\ \cup & \{100\} & \times\{B\} & \times\{A\} \\ \cup & \{101\} & \times\{B, C, D\} & \times\{A\} \\ \cup & \{110\} & \times\{B, D\} & \times\{B, C\} \\ \cup & \{111\} & \times\{B, C\} & \times\{B, C, D\} \\ \multicolumn{5}{l}{\quad \cup \{(110, B, D), (110, C, C), (111, D, B)\}\,.} \end{array}$$

It is an inductive invariant of the program. Note that this over-approximation doesn't contain a state of the form $(\_, D, D)$, so mutual exclusion is proven.

First thread waits

Let's try to prove that the label D never gets reached in the following program:

Global variable g=0

```
Thread 1:              ||  Thread 2:
A: wait until g=1;     ||  E: g:=1;
B: wait until g=0;     ||  F: g:=0;
C: wait until g=1;     ||  G:
D:                     ||
```

Let the local variable be the program counter pc ranging over $\{A, B, C, D, E, F, G\}$.

Choosing the empty exception set $E = \emptyset$ leads to $A = \mathrm{lfp}(\lambda X.\alpha_{\mathrm{mc}}(\mathrm{init} \cup \mathrm{post} \circ \gamma_{\mathrm{mc}}(X))) = (\{0A, 1A, 0B, 1B, 0C, 1C, 0D, 1D\}, \{0E, 1F, 0G\})$, whose concretization $\gamma_{\mathrm{mc}}(A)$ contains D as a local component of the first thread. Here, the simplified notation XY (e.g. $0A$) means the pair of valuation maps $(\{(g, X)\}, \{(pc, Y)\})$. The precision of the Flanagan-Qadeer algorithm (which is equivalent to our approach with the empty exception set) is insufficient.

For comparison we choose the exception set $E = \{0CG, 0BG\}$. (It is a shorthand for $\{(\{(g, 0)\}, \{(pc, C)\}, \{(pc, G)\}), (\{(g, 0)\}, \{(pc, B)\}, \{(pc, G)\})\}$, viewing each state as a tuple of valuation maps.) The initial state is $0AE$.

The fixpoint iteration sequence is $X_0 = (\emptyset, \emptyset)$ and $X_{i+1} = \alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init}) \sqcup \alpha_{\mathrm{mc}} \circ \alpha_E \circ \mathrm{post} \circ \gamma_E \circ \gamma_{\mathrm{mc}}(X^i)$ for $i \geq 0$:

$$
\begin{aligned}
X_0 &= (\emptyset, \emptyset) \\
X_1 &= (\{0A\}, \{0E\}) \sqcup \alpha_{\mathrm{mc}} \underbrace{((\mathrm{post}\{0CG, 0BG\}) \setminus \{0CG, 0BG\})}_{\emptyset} = (\{0A\}, \{0E\}) \\
X_2 &= (\{0A\}, \{0E\}) \sqcup \alpha_{\mathrm{mc}} \underbrace{((\mathrm{post}\{0BG, 0CG, 0AE\}) \setminus \{OBG, OCG\})}_{\{1AF\}} \\
&= (\{0A, 1A\}, \{0E, 1F\}) \\
X_3 &= (\{0A\}, \{0E\}) \sqcup \alpha_{\mathrm{mc}} \underbrace{((\mathrm{post}\{0BG, 0CG, 0AE, 1AF\}) \setminus \{OBG, OCG\})}_{\{1AF, 1BF, 0AG\}} \\
&= (\{0A, 1A, 1B\}, \{0E, 0G, 1F\}) \\
X_4 &= (\{0A\}, \{0E\}) \sqcup \alpha_{\mathrm{mc}} \underbrace{((\mathrm{post}\{0BG, 0CG, 0AE, 0AG, 1AF, 1BF\}) \setminus \{OBG, OCG\})}_{\{1AF, 1BF, 0AG\}} \\
&= (\{0A, 1A, 1B\}, \{0E, 0G, 1F\})
\end{aligned}
$$

$$\gamma_E \circ \gamma_{\mathrm{c}}(X_3) = \{0BG, 0CG, 0AE, 0AG, 1AF, 1BF\}\,.$$

The concretization of the fixpoint doesn't contain D as a local state of the first thread, which is what we wanted to prove.

Programs with locks

We show a suitable exception set for a class of programs with locks.

We describe now a class of multithreaded program with $n \geq 1$ threads and $k \geq 1$ locks. For simplicity, we number the lock variables by natural numbers. Let $\mathrm{Glob} = \mathfrak{P}(\mathbb{N}_k)$ be the set of global states. If a set $G \in \mathrm{Glob}$ is a part of a program state, then in this state exactly the locks in G are held. Let $\mathrm{Loc} = \mathbb{N}_L$ be the set of control locations of each thread for some $L \geq 1$. For simplicity, we assume no other local variables except the program counters and no other global variables except locks. The only allowed statements for any thread i are

- "j: acquire lck; j+1:", representing all transitions $(g, j) \to_i (g \cup \{\mathrm{lck}\}, j{+}1)$ where $\mathrm{lck} \notin g$;
- "j: release lck; $j+1$:"', representing all transitions $(g, j) \to_i (g \setminus \{\mathrm{lck}\}, j + 1)$;
- "j: if (ϕ) goto j'; j+1:", representing all transitions $(g, j) \to_i (g, j')$ where (g, j) satisfies the formula ϕ and all transitions $(g, j) \to_i (g, j + 1)$ where (g, j) doesn't satisfy ϕ.

Given the program text of a thread, a *critical section* is a contiguous sequence of control locations that starts right after a statement "acquire lck" and ends just before the next following "release lck" statement, for the same lock variable lck. If there is no following "release lck" statement, the critical section ends at the end of the thread. The lock variable "lck" is said to *protect* all the locations in this critical section. A program state has the *mutual exclusion* property if in this state, for every lock, at most one thread is in any of its critical sections protected by this lock. The whole program has the *mutual exclusion* property if there is no execution that starts in an initial state and ends in a state that violates mutual exclusion. The programs, for which we would like to derive the exception set, should satisfy the following criteria:

1. For any initial state, if a lock variable lck is held, then there is exactly one thread inside some of its critical sections protected by lck, and if it's not held, there is no thread inside any of its critical sections protected by lck.
2. The only jumps from inside a critical section to outside or vice versa are via the acquire and release statements.
3. Within any thread, for any $lck \in \mathbb{N}_k$ and statement "release lck", there is a preceding statement "acquire lck" without other "release lck" in between.

An example of a thread of such a program is

1: acquire lck_1;
2: acquire lck_2;
3: release lck_1;
4: if (lck_1) goto 7;
5: acquire lck_3;
6: release lck_3;
7:

The description of the program class is finished, now we are going to define the exception set for such a program.

An n-partition of a set is a tuple of n disjoint subsets whose union gives the set itself. Formally, an *n-partition* of a set X is a tuple $(X_i)_{i=1}^n \in (\mathfrak{P}(X))^n$ such that for all $i, j \in \mathbb{N}_n$ with $x \neq j$ we have $X_i \cap X_j = \emptyset$ and such that $\bigcup_{i=1}^n X_i = X$.

For $G \in \text{Glob}$, let $P_n(G)$ be the set of n-partitions of G. For $G \in \text{Glob}$ and $1 \leq i \leq n$, let $C(G,i) = \{j \in \text{Loc} \mid \forall\, lck \in \mathbb{N}_k : lck \text{ protects } j \text{ in thread } i \text{ iff } lck \in G\}$ be the set of control locations of thread i which are protected by exactly all the locks in G. For each n-partition $S = (S_i)_{1 \leq i \leq n}$ of a global store, let $C(S) = \prod_{i=1}^n C(S_i, i)$. For each $G \in \text{Glob}$, let $M(G) = \{C(S) \mid C(S) \neq \emptyset \text{ and } S \in P_n(G)\}$ and $E(G) = \bigcup M(G)$. Let the exception set be defined as $E = \bigcup_{G \in \text{Glob}, G \neq \emptyset} \{G\} \times E(G)$. Let $N = \{\emptyset\} \times E(\emptyset)$ and $R = E \cup N$.

Lemma 2.2.12. The set R is an inductive invariant.

Idea. Each n-partition of a shared state induces a distribution of locks on n threads. Each initial state fits into this distribution scheme and the transition relation doesn't break the scheme.

Proof. Let $(G, l) \in \text{init}$.

- (G, l) has no thread in critical section. By criterion 1, no lock is held. Thus $G = \emptyset$. So $(G, l) \in \{\emptyset\} \times \prod_{i=1}^n C(\emptyset, i) = \{\emptyset\} \times C((\emptyset)_{i=1}^n) = N \subseteq R$.
- (G, l) has some threads in critical section. By criterion 1, for each held lock, there is at most one thread in a critical section protected by this lock. Let $P(i) = \{lck \in G \mid lck \text{ protects } l_i \text{ in thread } i\}$ for all $i \in \mathbb{N}_n$. If we

have $P(i) \cap P(j) \neq \emptyset$ for some $1 \leq i, j \leq n$, $i \neq j$, then some $lck \in G$ would protect both l_i and l_j in threads i and j, respectively, in contradiction to criterion 1. So $P(i)$ and $P(j)$ are disjoint for different $1 \leq i, j \leq n$. By the same criterion, each $lck \in G$ protects l_i for some thread $1 \leq i \leq n$, implying $lck \in P(i)$. So $(P(i))_{i=1}^n$ is a n-partition of G. Now let $1 \leq i \leq n$. For all $lck \notin G$, neither $lck \in P(i)$ (since $P(i) \subseteq G$), nor lck protects l_i in thread i (by criterion 1). For all $lck \in G$, the fact $lck \in P(i)$ is equivalent to lck protecting l_i in thread i. So for all $lck \in \mathbb{N}_k$, the facts $lck \in P(i)$ and lck protects l_i in thread i are equivalent. This proves $l_i \in C(P(i), i)$. So $l \in C((P(i))_{i=1}^n) \subseteq E(G)$. Thus $(G, l) \in E \subseteq R$.

We have shown init $\subseteq R$.

Now let $(G, l) \in R$ with a successor (G', l'). Let i be the thread that contains the statement τ that induced the transition. Let S be a n-partition of G with $l \in C(S)$. Case split on τ.

- "l_i: acquire lck; l_i':". So $lck \notin G$ and $G' = G \cup \{lck\}$. Let $S_i' = S_i \dot\cup \{lck\}$ and $S_j' = S_j$ for $j \neq i$. Then $S' = (S_j)_{j=1}^n$ is a partition of G'. We have $l_i \in C(S_i, i)$, so for all $lck' \in \mathbb{N}_k \setminus \{lck\}$, we have that lck' protects l_i in thread i if and only if $lck' \in S_i$. Thus for all $lck' \in \mathbb{N}_k \setminus \{lck\}$, we have that lck' protects l_i' in thread i if and only if $lck' \in S_i'$. We also have that lck protects l_i' in thread i and $lck \in S_i'$. So l_i' in $C(S_i', i)$. Knowing that $l_j' = l_j \in C(S_j, j) = C(S_j', j)$ for $j \neq i$, we have $l' \in E(G')$ and $(G', l') \in E$.
- "l_i: release lck; l_i':". So $G' = G \setminus \{lck\}$. Let $S_i' = S_i \setminus \{lck\}$ and $S_j' = S_j$ for $j \neq i$. If $lck \in S_i$ then the tuple $S' = (S_j')_{j=1}^n$ is a partition of G'. If $lck \in S_j$ for some $j \neq i$, then $lck \notin S_i$, so $l_i \in C(S_i, i)$ implies that lck doesn't protect l_i in thread i, so there is no critical section for lck ending at l_i, thus "release lck" at l_i has no preceding "acquire lck", contradicting criterion 3. If $lck \notin S_j$ for any $j \in \mathbb{N}_n$, then $S' = S$ is a partition of $G' = G$. So in both cases $lck \in G$ and $lck \notin G$ the tuple S' is a partition of G'. We have $l_i \in C(S_i, i)$, so for all $lck' \in \mathbb{N}_k \setminus \{lck\}$ we have that lck' protects l_i in thread i if and only if $lck' \in S_i$. Thus for all $lck' \in \mathbb{N}_k \setminus \{lck\}$, we have that lck' protects l_i' in thread i if and only if $lck' \in S_i'$. Also neither lck protects l_i', nor $lck \in S_i'$. So $l_i' \in C(S_i', i)$. Knowing that $l_j' = l_j \in C(S_j, j) = C(S_j', j)$ for $j \neq i$, we have $l' \in E(G')$ and $(G', l') \in R$.
- "l_i: if(ϕ) goto b; c:". So $G = G'$. By criterion 2, no critical section is quit or entered by the conditional jump. So we have "$\forall\, lck \in \mathbb{N}_k : lck$ protects l_i in thread i" if and only if "$\forall\, lck \in \mathbb{N}_k : lck$ protects b in thread i" if and only if "$\forall\, lck \in \mathbb{N}_k : lck$ protects c in thread i". Thus $l_i \in C(S_i, i)$ implies $b, c \in C(S_i, i)$. Knowing that $l_i' \in \{b, c\}$ and $l_j' = l_j$ for $j \neq i$, we have $l' \in C(S)$ and thus $(G', l') \in R$.

□

Lemma 2.2.13. All states in R satisfy the mutual exclusion property.

Proof. Let $(G, l) \in R$ and $G \neq \emptyset$. Then $l \in C(S)$ for some n-partition of G, so $l_i \in C(S_i, i)$ for all $1 \leq i \leq n$. Let $lck \in \mathbb{N}_k$ and $i \neq j$ be two threads. If $lck \notin G$, then $lck \notin S_i \cup S_j$, so, by definition of C, the lock lck protects neither l_i nor l_j. If lck is in one of two sets S_i, S_j, say, in S_i, then $lck \notin S_j$, so lck protects l_i, but not l_j by definition of C. □

Notice that for $Y = (\{\emptyset\} \times C(\emptyset, i))_{i=1}^{n}$ we have $\gamma_{\mathrm{mc}}(Y) = N$, so applying Prop. 2.1.7 to $f = \lambda x.\, \mathrm{init} \cup \mathrm{post}(x)$, $R = \gamma_E \circ \gamma_{\mathrm{mc}}(Y)$ gives us $\gamma_E \circ \gamma_{\mathrm{mc}} \circ \mathrm{lfp}(\lambda x\, .\alpha_{\mathrm{mc}} \circ \alpha_E(\mathrm{init} \cup \mathrm{post} \circ \gamma_E \circ \gamma_{\mathrm{mc}}(x))) \subseteq R$. Thus doing thread-modular verification with exception set E proves mutual exclusion.

Lemma 2.2.14. For every $G \in \mathrm{Glob}$, the set $M(G)$ is a maximized antichain of Cartesian products.

Idea. Each location is protected by a fixed set of locks, implying incomparability. One proves the maximized property by taking a product inside the union, assuming it intersects at least two products in the antichain, and deriving a contradiction.

Proof. To show that $M(G)$ is an antichain, let S and S' be two different n-partitions of G such that $C(S) \neq \emptyset \neq C(S')$. Then there is some component $1 \leq i \leq n$ so that $S_i \neq S'_i$. Without loss of generality, let $lck \in S_i \setminus S'_i$ (if no such lck exists, swap S and S'). Assume that $j \in C(S_i, i) \cap C(S'_i, i)$. Since $j \in C(S_i, i)$ and $lck \in S_i$, the lock lck protects j in thread i. Since $j \in C(S'_i, i)$ and $lck \notin S'_i$, the lock lck doesn't protect j in thread i. This is a contradiction, so the assumption was false and $C(S_i, i) \cap C(S'_i, i) = \emptyset$. Together with $C(S_i, i) \neq \emptyset \neq C(S'_i, i)$ we obtain that $C(S)$ and $C(S')$ are incomparable with respect to product ordering.

To show that $M(G)$ is maximized, let's assume the contrary, namely that some Cartesian product $C = \prod_{i=1}^{n} C_i$ is a subset of $E(G)$, but not a subset of any $C(S) \neq \emptyset$ for an n-partition S of G. So let C have a nonempty intersection with $C(S)$ and a nonempty intersection with $C(S')$ for different $S, S' \in P_n(G)$. Let $a \in C \cap C(S)$ and $b \in C \cap C(S')$. There is a component $i \in \mathbb{N}_n$ in with $S_i \neq S'_i$. We have $a_i \in C(S_i, i)$, but $C(S_i, i)$ is disjoint from $C(S'_i, i)$, so $a_i \notin C(S'_i, i)$. But $b_i \in C(S'_i, i)$. Let $a' = (a \setminus \{(i, a_i)\}) \cup \{(i, b_i)\}$. Then $a'_i \in C_i$, so $a' \in C$. Knowing $S_i \neq S'_i$, there are two cases.

Case $S'_i \subsetneq S_i$. Let $d \in E(G)$. We claim:

$$\{lck \in \mathbb{N}_k \mid \exists\, j \in \mathbb{N}_n : lck \text{ protects } d_j \text{ in thread } j\} = G\,.$$

Let $lck \in \mathbb{N}_k$ be any lock that protects d_j in some thread $j \in \mathbb{N}_n$. There is a n-partition S'' of G with $d \in C(S'')$. Then $d_j \in C(S''_j, j)$. Thus $lck \in S''_j \subseteq G$. On the other hand, let $lck \in G$. There is a n-partition S'' of G with $d \in C(S'')$. There is a $1 \leq j \leq n$ with $lck \in S''_j$. We have $d_j \in C(S''_j, j)$. Then lck protects d_j in thread j. The claim is proven.

Knowing $a \in C \subset E(G)$, the components of a are protected by totally $|G|$ locks. From $S'_i \subsetneq S_i$, $a'_i = b_i \in C(S'_i, i)$ and $a_i \in C(S_i, i)$, we follow that the set of locks that protect a'_i in thread i is strictly smaller than the set of locks that protect a_i in thread i. For other threads $j \neq i$, the set of locks that protect $a_j = a'_j$ is S_j, which is disjoint from S_i. So a' is protected by less locks than a, thus, using our claim, $a' \notin E(G)$. It's a contradiction to $a' \in C \subseteq E(G)$.

Case $S'_i \setminus S_i \neq \emptyset$. Since S and S' both have G as a union, there is some component $j \neq i$ so that S_j contains at least one lock, say, lck, from $S'_i \setminus S_i$. So $a'_j = a_j \in C(S_j, j)$ is protected by lck. But $lck \in S'_i$ and $a'_i = b_i \in C(S'_i, i)$, so a'_i is protected by lck also. But any element $d \in E(G)$ has the property that for different components $\bar{i}, \bar{i}'$, the locations $d_{\bar{i}}$ and $d_{\bar{i}'}$ are protected by disjoint sets of locks. This property doesn't hold for a', so $a' \notin E(G)$, in contradiction to $a' \in C \subseteq E(G)$.

□

Lemma 2.2.15. Storing the exception set needs space which is exponential in k, polynomial in n and linear in L. More precisely:
On a RAM with logarithmic cost measure, the representation of the exception set E as a list of antichains of Cartesian products, one antichain per global part, needs space $O(nL(n+1)^k)$.

Proof. We have $|M(G)| \leq \sum_{k_1+\ldots+k_n=|G|} \binom{|G|}{k_1,\ldots,k_n} = n^{|G|}$. For storing a Cartesian product $C(S)$ for any partition S, we need Ln memory cells, storing each component as a bitvector, using one bit per addressed memory cell. The consumed space per Cartesian product is thus Ln. Storing $M(G)$ as an array needs $|M(G)|Ln + \log(|M(G)|) = n^{|G|+1}L + |G|\log n$ space. All such arrays, where each $G \in \text{Glob} \setminus \{\emptyset\}$ is stored again as an array, need the following space :

$$\begin{aligned}
&\sum_{G\in\text{Glob},G\neq\emptyset} (n^{|G|+1}L + \underbrace{|G|\log n}_{\leq n^{|G|}}) + \log(|\text{Glob}|-1) \;\leq \\
&\leq nL \sum_{\emptyset\neq G\subseteq\mathbb{N}_k} n^{|G|} + \sum_{\emptyset\neq G\subseteq\mathbb{N}_k} n^{|G|} + \log(2^k-1) \;\leq \\
&\leq nL((n+1)^k-1) + (n+1)^k - 1 + k \;\leq\; 3nL(n+1)^k\,.
\end{aligned}$$

□

Although the representation size is exponential in k, the number of global states, and thus the program size, is also exponential in k (with a different base).

For a constant number of locks, and a variable number of threads, the exception set still needs polynomial space, and thus allows computing the abstract fixpoint in polynomial time in the number of threads.

Chapter 3

Counterexample-Guided Abstraction Refinement

In the previous chapters we have shown how to use a fixed but arbitrary parameterization of thread-modular reasoning via an exception set. Now we are going to show how to find this parameterization.

3.1 Example

Now we sketch the main steps of a new verification algorithm on an example.

$$\textbf{shared } g = 0$$

$$P_1 :: \left[\begin{array}{l} \ell_1 : \textbf{acquire } g; \\ \ell_2 : \textbf{release } g; \\ \ell_3 : \end{array}\right] \quad \| \quad P_2 :: \left[\begin{array}{l} \ell_1 : \textbf{acquire } g; \\ \ell_2 : \textbf{release } g; \\ \ell_3 : \end{array}\right]$$

Figure 3.1: Example program.

The program in Fig. 3.1 has two threads. Each thread acquires a lock and immediately releases it. The operation "**acquire** g" atomically waits until g gets zero and sets g to one in the same transition, i.e. wait-and-set is executed atomically. The operation "**release** g" sets g to zero. We want to prove that whenever the program starts in the initial states

$$\text{init} = \{(0, \ell_1, \ell_2)\},$$

it never gets to the states in which both threads are in their critical sections at location ℓ_2 simultaneously. In fact, we want to prove that the unsafe states

$$\text{unsafe} = \{0, 1\} \times \{\ell_2\} \times \{\ell_2\}$$

are not reachable from the initial ones.

There is a reachable state of the program in which the first thread is in its critical section, for example $(1, \ell_2, \ell_1)$. In this state, the value of the shared

variable g is 1, the value of the program counter of the first thread is ℓ_2 and the value of the program counter of the second thread is ℓ_1. Symmetrically the same holds for the second thread: the state $(1, \ell_1, \ell_2)$ is also reachable. A thread-modular analysis forgets the dependencies between the threads, storing the information that the states of the first and second thread $(1, \ell_2)$ are reachable. For efficiency reasons, the thread-modular analysis would not maintain the combinations in which these thread states occurred. Nothing prohibits the thread-modular analysis from considering their combination $(1, \ell_2, \ell_2)$ as reachable.

On the abstract level, we would like to prevent the analysis from creating the combination $(1, \ell_2, \ell_2)$, but, for efficiency reasons, allow the analysis to create other combinations. To do that, a tuned thread-modular analysis should exempt such states from the recombination process that cause the creation of unsafe states, then execute the recombination and then add the exempted states back again. For our example it turns out that the following *exception set* suffices to prove correctness:

$$E = \{(1, \ell_2, \ell_1), (1, \ell_2, \ell_3)\}\,.$$

An exception set is subtracted before and added after a recombination. The algorithm that we will present will eventually find this exception set after trying out some candidates. The algorithm creates a monotonically growing sequence of state sets $(B_i)_{i\geq 1}$ such that B_1 contains the initial states and the successors of B_i are in B_{i+1} $(i \geq 1)$. Moreover, each B_i $(i \geq 1)$ is of the form

$$\{0\} \times \{\ldots\} \times \{\ldots\} \;\cup\; \{1\} \times \{\ldots\} \times \{\ldots\} \;\cup\; \text{ some exception set}\,. \qquad (3.1)$$

Here is this sequence (we will later show how it is derived):

$$\begin{aligned}
B_1 &= \{0\} \times \{\ell_1\} \times \{\ell_1\}\ (= \text{init})\,,\\
B_2 &= \{0\} \times \{\ell_1\} \times \{\ell_1\} \cup \{1\} \times \{\ell_1\} \times \{\ell_2\} \cup \{(1, \ell_2, \ell_1)\}\,,\\
B_3 &= \{0\} \times \{\ell_1, \ell_3\} \times \{\ell_1, \ell_3\} \cup \{1\} \times \{\ell_1\} \times \{\ell_2\} \cup \{(1, \ell_2, \ell_1)\}\,,\\
B_4 &= \{0\} \times \{\ell_1, \ell_3\} \times \{\ell_1, \ell_3\} \cup \{1\} \times \{\ell_1, \ell_3\} \times \{\ell_2\} \cup \{(1, \ell_2, \ell_1), (1, \ell_2, \ell_3)\}\\
&= B_5 = B_6 = \ldots
\end{aligned}$$

The ultimately stable exception set is the already mentioned E. The ultimately stable set B_4 is an inductive invariant sufficient to prove correctness.

Subsequently we show an algorithm that proves correctness by deriving such a sequence that ultimately stabilizes at an inductive invariant, represented in the form (3.1).

3.2 Counterexample-guided refinement loop for thread-modular verification

Now we show an algorithm that proves multithreaded program correctness. The algorithm derives a sequence of iterates and exception sets using counterexample-guided abstraction refinement.

Before describing the "lazy" CEGAR loop for thread-modular reasoning, we would like to remind the reader about the "lazy" abstraction for sequential programs inside the BLAST model checker [7]. In BLAST, an abstract reachability

tree is maintained, which represents states that are considered reachable from the initial one. A node of the tree represents a set of states that share the same program counter, an edge of a tree represents a statement. If some node is generated that has unsafe states, then a counterexample trace is computed from that node backwards towards the initial node. The earliest node from which an error trace starts, called the pivot, gets refined; the subtree rooted at the pivot might get thrown away and be recomputed with the new abstraction. "Laziness" in BLAST means that nodes before the pivot don't get refined.

In our algorithm the data structure is simpler: we use a chain instead of a tree. Each two consecutive nodes of the chain (we call these nodes iterates) are connected by an approximation of the transition relation of the program. In case some iterate intersects the error states, a trace backwards is computed until a pivot is found (i.e. the earliest iterate that leads to an error state via the found trace). The pivot is refined by tuning its abstraction, the subchain after that pivot is thrown away and gets recomputed again with the refined abstraction.

The refinement loop, depicted on Fig. 3.2, first computes the iterates with an approximation of the successor map post. In case a spurious counterexample (which is a sequence of iterates not fully contained in the safe states) gets found, the approximation of post gets refined.

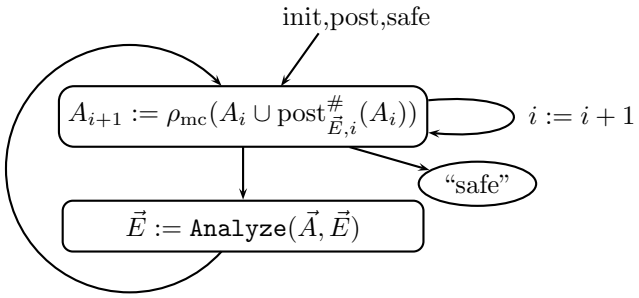


Figure 3.2: CEGAR-TM

The symbols $\vec{E} = (E_i)_{i\geq 1}$ and $\vec{A} = (A_i)_{i\geq 1}$ are sequences of sets.

The approximation of post is tuned by $\vec{E}$:

$$\mathrm{post}^{\#}_{\vec{E},i}(A_i) = \rho_{\mathrm{mc}}(\mathrm{post}(A_i \cup E_i) \setminus E_{i+1})\,.$$

The sets E_i $(i \geq 1)$ are exception sets: they get subtracted before the approximation happens and are added back before the next successor computation. Initially the exception sets are all empty.

Efficient computation of $\mathrm{post}^{\#}_{\vec{E},i}$ is a slight variation of the algorithm from Section 2.2. The algorithm of that section is formulated for the case that the exception set that is subtracted before approximation equals the exception set that is added before successor computation. The generalization to different exception sets goes along the lines of the previous proof, requiring no new ideas.

We call the pairs (A_i, E_i) *iterates* $(i \geq 1)$. If the sequence of iterates stabilizes at some i without touching an unsafe state, "safe" is reported. If an unsafe state is touched at some i, the so far created sequence of iterates is analyzed.

When we speak about an iterate (A_i, E_i) as about a set of states, we mean the union $A_i \cup E_i$; moreover, the *main* i^{th} iterate is A_i $(i \geq 1)$.

The procedure `Analyze`$(\vec{A}, \vec{E})$ looks for a counterexample in the current iterate sequence $(A_i, E_i)_{i \geq 1}$ by backtracking. If some is found, "unsafe" is reported. Otherwise the earliest iterate (pivot) at which a counterexample trace can start, is determined and used to generate a new sequence of exception sets. The procedure starts by assigning the intersection between the current iterate and the unsafe states to Bad and proceeding as in Fig. 3.3.

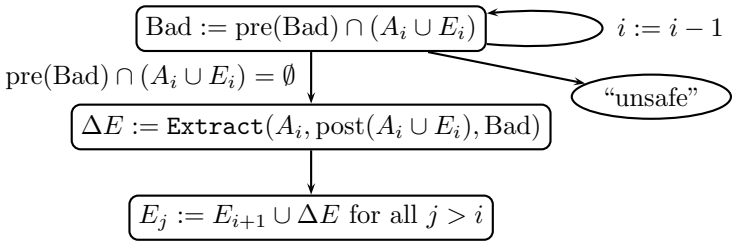


Figure 3.3: `Analyze`$(\vec{A}, \vec{E})$

Thus `Analyze` computes the traces from the unsafe states in the current iterate backwards to the initial state. If there is a trace starting in an initial state, "unsafe" is reported. Otherwise the algorithm determines the earliest iterate (pivot) from which there is a trace to the unsafe portion of the current iterate. That pivot had a too coarse approximation, so it and all subsequent iterates have to be recomputed. An exception set ΔE for the pivot and "bad states" Bad is computed by the procedure Extract. Then ΔE is added to the previous exception set for the pivot. This union is used for recomputation of all subsequent iterates.

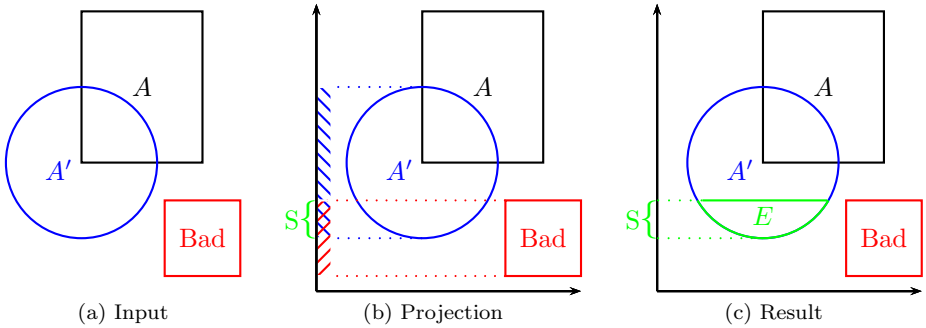
The pre- and the postcondition of the procedure Extract(A, A', Bad) are given in Fig. 3.4. These requirements ensure a correct and terminating analysis. Later we will show that the precondition is always satisfied.

Precondition: $A \in D^+$ and $(A \cup A') \cap \text{Bad} = \emptyset$.
Postcondition: returned $E \subseteq A'$ satisfies $\rho_{\text{mc}}(A \cup (A' \setminus E)) \cap \text{Bad} = \emptyset$.

Figure 3.4: Requirements for Extract(A, A', Bad)

Although taking A' as the result would satisfy the postcondition, it is inefficient in general, since A' may have a large representation. We are going to look at an optimized version. Our implementation of Extract finds an exception set for each shared part separately and then takes the union. So let's assume for simplicity that the shared part is a singleton. So let A be a product of n sets of local states, A' and Bad be any sets of n-tuples and $(A \cup A') \cap \text{Bad} = \emptyset$. The implementation proceeds by looking at the representation of Bad, which is in our case represented as a union of Cartesian products. Then the implementation solves the problem for each product in the representation of Bad separately, and then takes the union. So let's assume for simplicity that Bad is a Cartesian product. Such a typical situation is shown in Fig. 3.5a.

Since both A and Bad are disjoint Cartesian products, there is some dimension in which their component sets are disjoint. In the situation on Fig. 3.5a, the projections of A and Bad on the horizontal axis are intersecting, while the projections of A and Bad on the vertical axis are disjoint. Let's look at the pro-



(a) Input (b) Projection (c) Result

Figure 3.5: Extract(A, A', Bad)

jections of A' and Bad on the vertical axis. These projections have a common subset, denoted on Fig. 3.5b by S. The implementation returns the set of those points of A' whose vertical component is in S. In the situation on Fig. 3.5a, Bad intersects the Cartesian approximation of $A \cup A'$, but does not intersect the Cartesian approximation of $A \cup (A' \setminus E)$ on Fig. 3.5c.

In all described computations, we use Cartesian products and their unions to represent sets of n-tuples. In general, any set can be represented in different ways by unions of products. We optimize towards a representation with a small number of products. This is done by merging two products into one whenever no precision gets lost.

We state now and will show later the following properties of the algorithm.

Correctness. If the algorithm returns "safe", then the property holds. If the algorithm returns "unsafe", then the property is violated.

A *counterexample* in our context is just a sequence of abstract iterates together with the corresponding exception sets that starts with (A_1, E_1) and ends in an iterate (A_i, E_i) such that $A_i \cup E_i$ intersects the unsafe states. The algorithm makes progress:

Progress. Every counterexample gets eliminated, i.e. no counterexample occurs twice in a run of the refinement loop.

The reason behind progress is that the iterates sequences decrease strictly lexicographically from one refinement phase (which is a maximal segment of a run of CEGAR-TM with a fixed sequence of exception sets $\vec{E}$) to the next one. Formally speaking, whenever $(A_i \cup E_i)_{i \geq 1}$ and $(\widetilde{A_i} \cup \widetilde{E_i})_{i \geq 1}$ are iterates sequences from some fixed and the following refinement phases, respectively, then

$$(A_i \cup E_i)_{i \geq 1} \supsetneq_{\text{lex}} (\widetilde{A_i} \cup \widetilde{E_i})_{i \geq 1} .$$

It means that both sequences have a common (maybe empty) prefix and in the earliest component i in which they differ we have $A_i \cup E_i \supsetneq \widetilde{A_i} \cup \widetilde{E_i}$. This progress property holds for the refinement scheme even if the programs are infinite-state (and all steps of the refinement loop are still computable, for example due to bounded branching of the transition relation and a finite number of initial states).

We may even claim a stronger progress property by considering the so-called distances from the iterates to unsafe states. For that, let ∞ be any infinite ordinal and the set $\mathbb{N}_0 \dot{\cup} \{\infty\}$ be equipped with the usual order on the ordinals.

Let the *distance* from a sets of states A to a set of states B is

$$d(A, B) = \min\{j \in \mathbb{N}_0 \mid \text{post}^j(A) \cap B \neq \emptyset\},$$

and the distance of the i^{th} iterate to unsafe is

$$d_i = d(A_i \cup E_i, \text{unsafe}),$$

with the convention that the minimum of the empty set is ∞.

One can show that the sequence of distances from iterates to error states increases strictly lexicographically from one refinement phase to the next one. This implies termination and an upper bound on the runtime.

Termination. The algorithm terminates within 2^{gl^n} refinement phases, where $g = |\text{Glob}|$ and $l = |\text{Loc}|$.

For properties which don't hold we get a better worst-case bound:

Termination on safety. If the property doesn't hold, then the algorithm returns "unsafe". Further, if the distance from init to unsafe is $m \in \mathbb{N}_0$, then the algorithm terminates within 2^m refinement phases.

We say that a safety property is *inductive* if it is closed under the program's transition relation. For such properties a speedup can be achieved.

Fast termination. Let the property be inductive. Then the algorithm terminates within $gl^n + 1$ refinement phases. If the property does not hold, the algorithm terminates in the initial refinement phase at the first iterate.

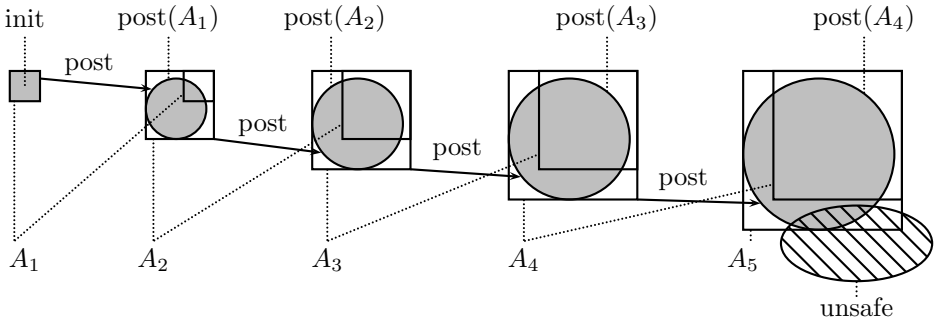
The bound is linear in the size of the transition system. We are not aware of linear bounds (in the size of the transition system) on the number of refinement phases for other CEGAR algorithms that start with a fixed approximation and refine it subsequently.

For a practically interesting class of programs we will show a better polynomial-time behavior later.

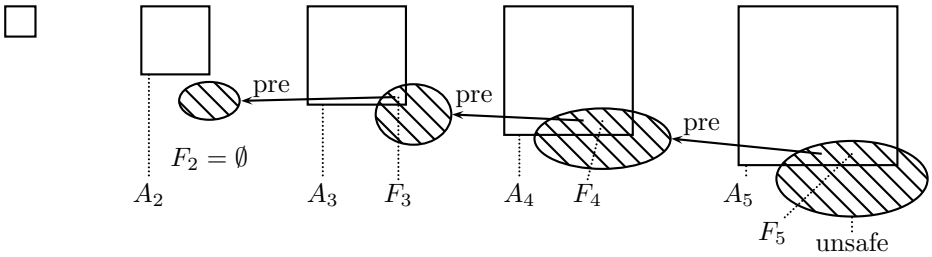
3.3 A typical scenario

The following pictures visualize a common scenario of how the algorithm works.

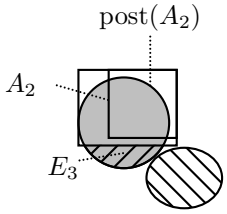
Assume for simplicity that there is exactly one shared state. The algorithm starts with the empty exception set, the iterate A_5 is the earliest iterate that touches an error state.



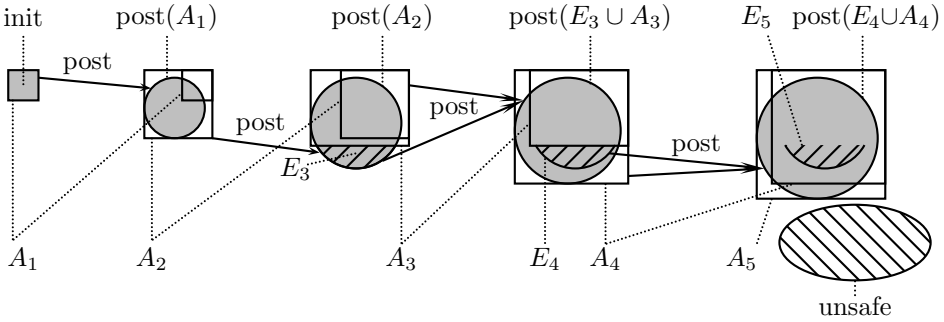
Now the ancestors of the states in the intersection of the error states with A_5 are computed:



Now a nonempty exception set $E_3 = E_4 = \ldots$ is created:



After recomputing A_3 we obtain a new smaller iterate number 3 and recompute all iterates from number 3 onwards:



The spurious counterexample (namely the sequence of iterates of the previous refinement phase) has been eliminated. As we will show later, we can guarantee that there is no two-step real trace from the states of newly computed third iterate to error states. If (A_5, E_5) is the fixed point, then safety is proven. Otherwise additional iterations are needed, which can in turn lead to refinements of any iterate.

3.4 Example revisited

Now we show how for the example of Fig. 3.1 the algorithm arrives at an inductive invariant that is sufficient to prove correctness.

3.4.1 Trying the empty exception set

We start with the initial states init $= \{0\} \times \{\ell_1\} \times \{\ell_1\}$, i.e. the value of g is zero and the threads are at their initial locations. The multithreaded Cartesian approximation ρ_{mc} takes for each shared part (here only 0) the set of corresponding local parts (here $\{\ell_1\} \times \{\ell_1\}$) and applies Cartesian approximation to this set of local parts. Applying Cartesian approximation means taking the smallest superset that is a Cartesian product. In our case the approximation

doesn't lose precision:

$$A_1 := \{0\} \times \{\ell_1\} \times \{\ell_1\}\,,$$

and the mutual exclusion property is yet satisfied: $A_1 \cap \text{unsafe} = \emptyset$. Taking successors produces

$$A_1' := \text{post}(A_1) = \{1\} \times \left(\begin{array}{l} \{\ell_2\} \times \{\ell_1\} \cup \\ \{\ell_1\} \times \{\ell_2\} \end{array} \right),$$

The multithreaded Cartesian approximation of this set is

$$\rho_{\text{mc}}(A_1') = \{1\} \times \{\ell_1, \ell_2\} \times \{\ell_1, \ell_2\}\,.$$

Now we take the componentwise union (on locals that belong to the same valuation of g) with the previous set:

$$A_2 := \rho_{\text{mc}}(A_1 \cup \rho_{\text{mc}}(A_1')) = \{0\} \times \{\ell_1\} \times \{\ell_1\} \cup \{1\} \times \{\ell_1, \ell_2\} \times \{\ell_1, \ell_2\}\,.$$

The set A_2 violates the mutual exclusion property:

$$\text{Bad}_2 := A_2 \cap \text{unsafe} = \{1\} \times \{\ell_2\} \times \{\ell_2\}$$

is not empty.

In Fig. 3.6 we see iterates for shared states 0 and 1 separately.

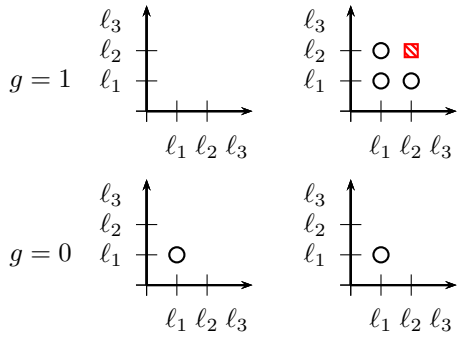


Figure 3.6: Sequence of iterates for the empty exception set.

3.4.2 Analyzing the failed proof attempt

Let's look at what `Analyze` does. We want to find the computation step at which too much precision was lost. So we compute the predecessors of Bad_2 that are in A_1:

$$\text{Bad}_1 := \text{pre}(\text{Bad}_2) \cap A_1 = \emptyset\,.$$

We didn't find a trace from the initial states. So we refine the second iterate by first taking away some set of states out of A_1', then applying the approximation ρ_{mc}, and the adding this set of states back. That is, we want an exception set $E \subseteq A_1'$ such that $\text{Bad}_2 \cap \rho_{\text{mc}}(A_1 \cup \rho_{\text{mc}}(A_1' \setminus E)) = \emptyset$. This E is generated by $\text{Extract}(A_1, A_1', \text{Bad}_2)$ by finding exception sets for each shared part separately and then taking the union.

For shared part 0 we can take the empty exception set, since A'_1 doesn't contain states with shared part 0.

For shared part 1, the local parts of the corresponding sets are $\widetilde{A_1} := \emptyset$, $\widetilde{A'_1} := \{\ell_2\} \times \{\ell_1\} \cup \{\ell_1\} \times \{\ell_2\}$ and $\widetilde{\text{Bad}_2} := \{\ell_2\} \times \{\ell_2\}$. First we find a dimension in which $\widetilde{A_1}$ and Bad_2 are disjoint. In our case, both dimensions work. We choose, say, dimension 1. The projection of $\widetilde{A'_1}$ on the first dimension is $\{\ell_1, \ell_2\}$, the projection of $\widetilde{\text{Bad}_2}$ on the first dimension is $\{\ell_2\}$. The intersection of both projections is $\{\ell_2\}$, put those states of $\widetilde{A'_1}$ into the exception set $\widetilde{E}$ that have $\{\ell_2\}$ as the local component of the first thread. So let $\widetilde{E} := \{\ell_2\} \times \{\ell_1\}$.

Combining the empty exception set for $g = 0$ and $\widetilde{E}$ for $g = 1$ as in Fig. 3.7 we set

$$E_2 := E_3 := E_4 := E_5 = \ldots = \{1\} \times \{\ell_2\} \times \{\ell_1\}\,.$$

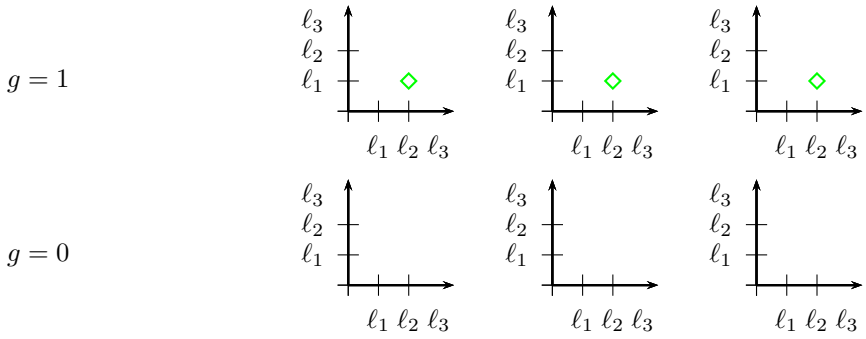


Figure 3.7: The first exception set.

The exception set for the first iterate remains unchanged, i.e. in our case empty.

3.4.3 Trying the found exception set

Now we recompute the iterates lazily from the point at which the too coarse approximation happened, i.e. from the second iterate onwards:

$$A_2 \;:=\; \rho_{\text{mc}}(A_1 \cup \rho_{\text{mc}}(A'_1 \setminus E_2)) \;=\; \{0\} \times \{\ell_1\} \times \{\ell_1\} \cup \{1\} \times \{\ell_1\} \times \{\ell_2\}\,.$$

These states are still safe: $(E_2 \cup A_2) \cap \text{unsafe} = \emptyset$. The successors are

$$A'_2 \;:=\; \text{post}(E_2 \cup A_2) \;=\; \{0\}\times \left(\begin{array}{l} \{\ell_3\}\times\{\ell_1\} \cup \\ \{\ell_1\}\times\{\ell_3\} \end{array} \right) \cup \{1\}\times \left(\begin{array}{l} \{\ell_2\}\times\{\ell_1\} \cup \\ \{\ell_1\}\times\{\ell_2\} \end{array} \right).$$

Now we approximate this set of states using the same exception set:

$$A_3 \;:=\; \rho_{\text{mc}}(A_2 \cup \rho_{\text{mc}}(A'_2 \setminus E_3)) \;=\; \{0\}\times\{\ell_1,\ell_3\}\times\{\ell_1,\ell_3\} \;\cup\; \{1\}\times\{\ell_1\}\times\{\ell_2\}\,.$$

Still no unsafe states are touched. Let's compute the successors once again:

$$A'_3 \;:=\; \text{post}(E_3 \cup A_3) \;=\; \{0\}\times \left(\begin{array}{l} \{\ell_1\}\times\{\ell_3\} \cup \\ \{\ell_3\}\times\{\ell_1\} \end{array} \right) \cup \{1\}\times \left(\begin{array}{l} \{\ell_2\}\times\{\ell_1,\ell_3\} \cup \\ \{\ell_1,\ell_3\}\times\{\ell_2\} \end{array} \right).$$

Computing the next main iterate gives

$$A_4 := \rho_{\mathrm{mc}}(A_3 \cup \rho_{\mathrm{mc}}(A_3' \setminus E_4)) \;=\; \{0\}\times\{\ell_1,\ell_3\}\times\{\ell_1,\ell_3\} \cup \{1\}\times\{\ell_1,\ell_2,\ell_3\}\times\{\ell_2,\ell_3\}\,.$$

Now the unsafe states are touched once again:

$$\mathrm{Bad}_4 \;:=\; (E_4 \cup A_4) \cap \mathrm{unsafe} \;=\; \{1\} \times \{\ell_2\} \times \{\ell_2\}\,.$$

In Fig. 3.8 we see the generated sequence of iterates.

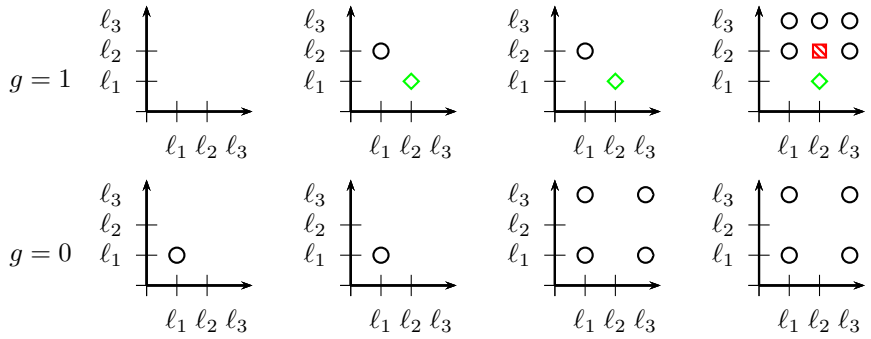


Figure 3.8: Sequence of iterates in the next refinement phase.

3.4.4 Analyzing the second proof failure

Now `Analyze` backtracks to see where the too coarse approximation has happened:

$$\mathrm{Bad}_3 \;:=\; \mathrm{pre}(\mathrm{Bad}_4) \cap (E_3 \cup A_3) \;=\; \emptyset\,.$$

So the earliest discovered too coarse approximation happened at iterate 4. We have $\mathrm{Bad}_4 \cap (A_3 \cup A_3') = \emptyset$ and $\mathrm{Bad}_4 \cap \rho_{\mathrm{mc}}(A_3 \cup \rho_{\mathrm{mc}}(A_3')) \neq \emptyset$. The procedure Extract should produce a set $E \subseteq A_3'$ such that $\mathrm{Bad}_4 \cap \rho_{\mathrm{mc}}(A_3 \cup \rho_{\mathrm{mc}}(A_3' \setminus E)) = \emptyset$. As before, let's extract the exception set for each shared part separately.

The set Bad_4 has no states with shared part 0, so the empty exception set for shared part 0 suffices.

For shared part 1 consider the corresponding local parts $\widetilde{A_3} := \{\ell_1\}\times\{\ell_2\}$, $\widetilde{A_3'} := \{\ell_2\}\times\{\ell_1,\ell_3\} \cup \{\ell_1,\ell_3\}\times\{\ell_2\}$ and $\widetilde{\mathrm{Bad}_4} := \{\ell_2\}\times\{\ell_2\}$. First we have to choose a thread such that $\widetilde{\mathrm{Bad}_4}$ has no common local states of that thread with $\widetilde{A_3}$. This is thread number 1: the projections of $\widetilde{A_3}$ and $\widetilde{\mathrm{Bad}_4}$ on the first dimension are disjoint. Now put those elements of $\widetilde{A_3'}$ that have the common first component with $\widetilde{\mathrm{Bad}_4}$ into the exception set: $\widetilde{E} := \{\ell_2\}\times\{\ell_1,\ell_3\}$.

Combining the exception sets for shared parts 0 and 1 and taking the union with the previous exception set E_4 as in Fig. 3.9 results in

$$E_4 \;:=\; E_5 \;:=\; E_6 \;:=\; \ldots \;=\; \{1\} \times \{\ell_2\} \times \{\ell_1,\ell_3\}\,.$$

The exception sets and consequently the iterates before the fourth remain as before.

Figure 3.9: Next exception set.

3.4.5 Succeeding with the second choice of the exception set

Now we recompute the iterates from the fourth onwards.

$$A_4 \;:=\; \rho_{\mathrm{mc}}(A_3 \cup \rho_{\mathrm{mc}}(A_3' \setminus E_4)) \;=\; \{0\}\times\{\ell_1,\ell_3\}\times\{\ell_1,\ell_3\} \cup \{1\}\times\{\ell_1,\ell_3\}\times\{\ell_2\}\,.$$

The fourth iterate is safe: $(A_4 \cup E_4) \cap \mathrm{unsafe} = \emptyset$. In Fig. 3.10 we see the generated sequence of iterates.

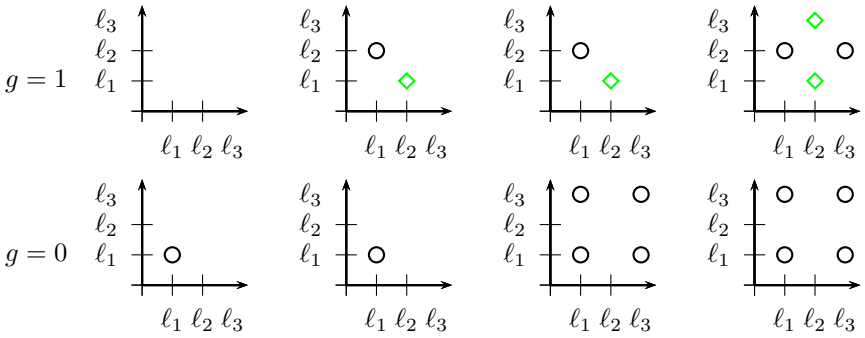


Figure 3.10: Sequence of iterates in the last refinement phase.

The successors of the fourth iterate are:

$$A_4' \;:=\; \mathrm{post}(E_4 \cup A_4) \;=\; \{0\}\times\left(\begin{array}{r}\{\ell_1,\ell_3\}\times\{\ell_3\}\cup\\ \{\ell_3\}\times\{\ell_1,\ell_3\}\end{array}\right) \cup \{1\}\times\left(\begin{array}{r}\{\ell_1,\ell_3\}\times\{\ell_2\}\cup\\ \{\ell_2\}\times\{\ell_1,\ell_3\}\end{array}\right).$$

The next main iterate is then

$$A_5 \;:=\; \rho_{\mathrm{mc}}(A_4 \cup \rho_{\mathrm{mc}}(A_4' \setminus E_5)) \;=\; \{0\}\times\{\ell_1,\ell_3\}\times\{\ell_1,\ell_3\} \cup \{1\}\times\{\ell_1,\ell_3\}\times\{\ell_2\}\,.$$

Since $A_4 = A_5$ and $E_4 = E_5$, the computation has stabilized and will not discover new states. The inductive invariant $E_4 \cup A_4$ proves correctness.

3.5 Properties*

Now we are going to formulate and prove the properties stated in Section 3.2. This section may be skipped during first reading.

In the above presentation of CEGAR-TM we have deliberately not taken care of details that can be readily derived (the checks for safety and unsafety, statement labels, etc). In the following lemmas we remain flexible with respect to such implementation details of CEGAR-TM; the proof sketches can certainly be written out in full length for a particular implementation of the algorithm, e.g. as in [11]. After each numbered lemma or theorem we show its proof sketch.

The proofs assume a run of CEGAR-TM on an arbitrary but fixed multi-threaded program.

Lemma 3.5.1 (Monotonically increasing sequences)**.** For a fixed refinement phase the following claims hold.
(a) The sequence of exception sets $(E_j)_{j\in\mathbb{N}^+}$ is monotonically increasing.
(b) The sequence of iterates $(A_j \cup E_j)_{j\in\mathbb{N}^+}$ is monotonically increasing.
(c) The sequence of iterates $(A_j \cup E_j)_{j\in\mathbb{N}^+}$ is pointwise greater than or equal to $((\lambda x.\, \mathrm{init} \cup \mathrm{post}(x))^j(\emptyset))_{j\in\mathbb{N}^+}$.

The lemma is proven by induction over the lexicographic order on the pairs (refinement phase, iterate number).

Lemma 3.5.2. For all refinement phases, all main iterates A_j ($j \in \mathbb{N}^+$) are in the image of ρ_{mc}.

The lemma is proven by induction on iterate numbers.

Lemma 3.5.3 (Coinciding exception sets and abstract elements)**.** Fix a refinement phase. Then all iterates (A_j, E_j) (and consequently their successors $\mathrm{post}(A_i \cup E_i)$) before the pivot maintain their values from the current to the next refinement phase.

The lemma is proven by induction on the iterate numbers.

Lemma 3.5.4. Fix a refinement phase and an iterate number $j \geq 1$. Then $E_{j+1} \subseteq E_j \cup \mathrm{post}(A_j \cup E_j)$.

Follows by induction on the refinement phase numbers, using Lemma 3.5.3 and postcondition of Extract.

We say that a refinement phase is *full* if the pivot has been determined in that phase.

Lemma 3.5.5. Fix a full refinement phase. Let $j \geq 1$ be the number of the pivot. Then $\mathrm{Bad}_j \cap ((E_{j-1} \cup A_{j-1}) \cup \mathrm{post}(E_{j-1} \cup A_{j-1})) = \emptyset$.

If the claim would not hold then either the unsafe states would get touched one iterate earlier or the refined iterate would be $\leq j-1$ instead of j.

Proposition 3.5.6. The precondition of Extract always holds.

The proposition follows from Lemmas 3.5.2, 3.5.4 and 3.5.5.

A consequence of the proposition is that the algorithm is well-defined.

Lemma 3.5.7. Fix a full refinement phase. Let $j \geq 1$ be the pivot iterate. Then $\mathrm{Bad}_j \cap (E_j \cup A_j) \neq \emptyset$. Moreover, if there is a next refinement phase, then E_j strictly grows from the current to the next refinement phase.

The first part of the lemma follows from the definition of `Analyze`; assuming that the exception set E_j remains the same leads to a contradiction to the first part.

Theorem 3.5.8 (Correctness)**.** Assume the algorithm terminates.
1. If the algorithm returns "safe", then the property holds.
2. If the algorithm returns "unsafe", then the property is violated.

1. If there is only one refinement phase, then as long as the exception set remains empty, safety is guaranteed by Lemma 3.5.1(b),(c) and Kleene's fixpoint theorem for fixpoints of join-morphisms.
   Otherwise fix the last refinement phase. Let last be the iterate number with $(A_{\text{last}-1}, E_{\text{last}-1}) = (A_{\text{last}}, E_{\text{last}})$ of the previous refinement phase. Notice that all the other iterates of the previous refinement phase are pairwise different. Lemmas 3.5.3 as well as Lemmas 3.5.1(a) and 3.5.7 show that the iterates of the last refinement phase strictly grow until the previous pivot. Thus the iterate at which "safe" is returned, is past the previous pivot. Notice that the exception set sequence ultimately stabilizes. Lemma 3.5.1(b),(c) and Kleene's fixpoint theorem imply safety.
2. `Analyze` constructs a sequence of bad regions for each iterate. In this sequence there is a trace from the initial states (in the very first iterate) till the unsafe states in the last constructed iterate.

Theorem 3.5.9 (Progress)**.** Consider any two adjacent refinement phases. The later sequences of iterates is strictly lexicographically smaller than the former sequence of iterates. The sequences start to differ at a pivot of the earlier refinement phase.

Lemma 3.5.3 and the postcondition of Extract imply that the later iterate sequence is smaller than or equal to the earlier one. Inequality follows from Lemma 3.5.7.

Lemma 3.5.10 (Strictly increasing iterate sequence)**.** In each refinement phase the sequence of iterates is strictly monotonically increasing either indefinitely or until it gets stationary.

Monotonicity follows from Lemma 3.5.1(b). Each iterate is different from the next one, either the main part has strictly increased or the exception set has strictly increased (or both). If the union would remain the same, we would obtain a contradiction for an earlier refinement phase between Lemmas 3.5.7 and 3.5.5 using Lemmas 3.5.3, 3.5.1(a) and 3.5.4.

Lemma 3.5.11. For any nonnegative m the set of m^{th} successors of the initial states is included into the set of the $m+1$st successors of the empty set with respect to the operator $\lambda x.\text{init} \cup \text{post}(x)$. The least fixpoint of this operator is given by the union over the all j^{th} successors of the initial states ($j \geq 0$).

For the proof, use induction and Kleene's fixpoint theorem.

Lemma 3.5.12. For any refinement phase, successors of the set represented by an iterate are included in the set represented by the next iterate.

Lemma follows from the defining equation of the iterates.

Lemma 3.5.13. Fix a refinement phase. The sequence of distances to the error states is either constantly ∞, or first ∞ for a finite (maybe empty) prefix, then strictly decreasing, and at last constant zero.

Follows from Lemma 3.5.12.

Lemma 3.5.14 (Bound on the last iterate number)**.** If the program is finite-state, for each refinement phase the number of generated iterates doesn't exceed the number of states. If the program is unsafe, for each refinement phase the number of generated iterates doesn't exceed the distance from initial to error states plus one.

If the program is finite-state, Lemma 3.5.10 bounds the length of the strictly ascending part of the sequence of iterates. If the distance from initial to error states is finite, Lemma 3.5.11 implies that that no more iterates are generated than the distance plus one.

Theorem 3.5.15 (Termination on unsafety)**.** If the property doesn't hold, then the algorithm returns "unsafe". Further, if the distance from init to unsafe is $m \in \mathbb{N}_0$, then the algorithm terminates within 2^m refinement phases.

Let a_j be the distance from the j^{th} iterate $A_j \cup E_j$ to the error states. Consider the sequence $(a_j)_{j \in \mathbb{N}^+}$. From one refinement round to the next one, the distance sequence prefix remains unchanged till the pivot iterate number and at the pivot iterate number the distance increases. Thus two different refinement phases have different sequences, the later one is lexicographically greater. Each distance sequence starts with m. Lemma 3.5.13 shows that each of these sequences strictly decreases down to zero. The number of such sequences is at most 2^m.

Theorem 3.5.16 (Termination for the finite-state case)**.** For a finite-state program, the algorithm terminates within 2^{gl^n} refinement phases.

If the property doesn't hold, termination follows from Thm. 3.5.15. Otherwise apply Thm. 3.5.9 and bound the number of iterates sequences of the form of Lemma 3.5.10.

Lemma 3.5.17. Let the property be inductive. Then for each refinement phase i, if there is a pivot, it is the very last iterate. If there is a next refinement phase $i+1$, it has at least as many iterates. If the next refinement phase $i+1$ has a pivot, it has strictly more iterates.

Since the property is inductive, the predecessors of error states are error states themselves. So backward search is never needed. Thus for two adjacent refinement phases, if the error states are not found in the latter refinement phase, then the latter refinement phase still generates at least as many iterates as the former one. If the error states are found in the latter refinement phase, then they are found at a strictly larger distance.

Theorem 3.5.18 (Fast termination for inductive properties)**.** Let the property be inductive. If the program is finite-state, the algorithm terminates within $gl^n + 1$ refinement phases. If the property does not hold, the algorithm terminates in the initial refinement phase at the first iterate.

The sequence of iterates strictly increases in length from one phase to the next one by Lemma 3.5.17 until the last phase, in which it may retain its length (in case of safety) or be otherwise longer. So the number of refinement phases is bounded by the length of a longest ascending sequence of iterates (in case of safety one additional refinement phase is needed that doesn't find any error state). Apply Lemma 3.5.14. If the inductive property does not hold, then already the initial states violate the property, causing the algorithm to terminate immediately.

3.6 Programs with locks

Now we describe a class of programs that can be automatically verified in polynomial time.

Each program in the class is generated by instantiating the schema shown in Figure 3.11 with a fixed number n of threads, a fixed number m of critical section per thread such that each critical section has k control locations.

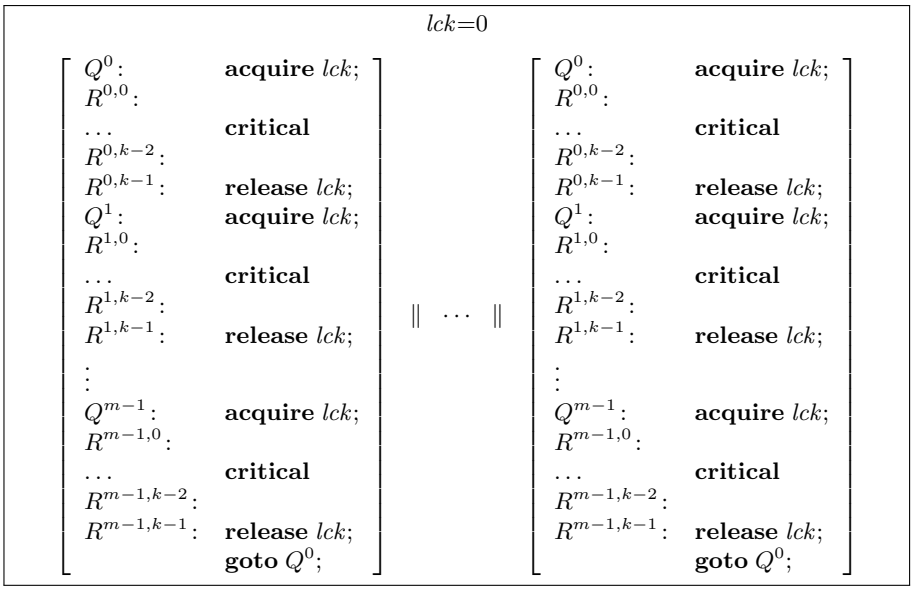


Figure 3.11: Schema for programs consisting of n concurrent threads with m critical sections per thread such that each critical section has k control locations. This class admits polynomial, efficient, precise and automatic thread-modular verification.

We remind that, since *lck* is boolean, this class has no thread-modular proof.

Let $C = \{R^{j,l} \mid j < m \text{ and } l < k\}$ be the critical local states, $N = \{Q^j \mid j < m\}$ the noncritical local states and $\text{Loc} = C \dot\cup N$ the local states of a thread. The error states are

$$\{0,1\} \times \{(a_i)_{i=1}^n \in \text{Loc}^n \mid \exists\, i,j \in \mathbb{N}_n : i \neq j \text{ and } a_i \in C \text{ and } a_j \in C\}\,.$$

For the proof of polynomial complexity we choose an eager version of generating an exception set. This means that in Fig. 3.4, not only one axis is taken

with disjoint projections of A and Bad, but all such axes. For each such axis an exception set is computed; then the union over just generated sets is produced.

The CEGAR algorithm needs mk refinement phases. In each phase, a new critical location is discovered. More specifically, in phases jk ($j < m$), the locations Q^j and $R^{j,0}$ are discovered. In phases $jk + r$ ($j < m$ and $1 \leq r < k$), the location $R^{j,r}$ is discovered. In each phase at least one new location is discovered because the set of error states has no predecessors and backtracking is not necessary. At the same time, no more than one critical location per phase is discovered: due to symmetry, when a new critical location of one thread is discovered, so it happens for all the threads. Since this critical location is new, it is not in the current exception set, thus it gets subjected to Cartesian abstraction, which leads to tuples with more than one critical location (because of symmetry). Then the error states are hit and the exception set is enlarged. The eager version of generating the exception set `Extract` produces, simplifying, all tuples where one component is critical and might include the new location (and the critical locations of the previous critical sections) and the other components are noncritical. This new exception set turns out to be equal to the current set of successors in their critical sections (if it were not, the difference between the successors and the exception set had at least one critical location, and, by symmetry, at least n, which would lead to error states after approximation). Subtracting the exception set from the successor set produces only tuples of noncritical locations, which get abstracted to a product of noncritical locations.

It is not immediately clear that each exception set has a polynomial maximized form. Instead of providing page-long listings as in [11] for a specific algorithm variant, we just give the central computation result, namely the exception set for each phase $jk + r$ (($j < m$ and $r < k$) or ($j = m$ and $r = 0$)). We are interested in asymptotic behavior and thus show the derived exception set for large parameter values $n \geq 3$, $m \geq 1$ and $k \geq 2$ (for smaller values the exception sets are simpler).

Let $B(U, V)$ be the union over n-dimensional products in which exactly exactly one component set is V and the remaining are U. Now
$E_1 = \emptyset$ and
$E_{p(k+1)+2+l} = \{1\} \times (B(\{Q^{p'} \mid p' < p\}, \{R^{p',l'} \mid p' < p \wedge l' < k\}) \cup B(\{Q^{p'} \mid p' \leq p\}, \{R^{p',l'} \mid p' \leq p \wedge l' \leq \min\{l, k-1\}\}))$, whose maximized form is
$\{1\} \times (B(\{Q^{p'} \mid p' < p\}, \{R^{p',l'} \mid (p' < p \wedge l' < k) \vee (p' \leq p \wedge l' \leq \min\{l, k-1\})\}) \cup B(\{Q^{p'} \mid p' \leq p\}, \{R^{p',l'} \mid p' \leq p \wedge l' \leq \min\{l, k-1\}\}))$ for $p < j$ and $l \leq k$ as well as for $p = j$ and $l < r$,
the ultimate exception set is $E_{j(k+1)+1+r}$.

This representation is maximized: if some product is a subset of restriction of $E_{p(k+1)+2+l}$ to shared part 0 (resp. to shared part 1), it is a subset of some of the products in the above representation for shared part 0 (resp. for shared part 1). By Subsection 2.2.3, each phase takes polynomial computation time. Since the number of phases is polynomial, the whole run of the algorithm takes polynomial time.

In Fig. 3.12, where the experimental results are depicted, a number in the parentheses denotes the number of critical sections m. The number of critical locations per critical section is 1. The curve denoted by the star ($\star$) depicts the time of a non-modular plain model-checking procedure without abstraction for an n-threaded program with one critical section per thread.

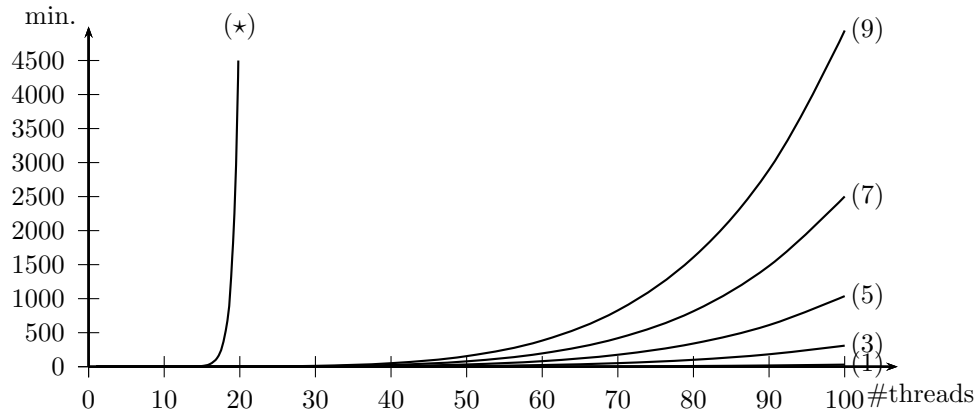


Figure 3.12: Non-modular ($\star$) vs. CEGAR-TM (number of critical sections) verification with exception sets. The locks class is scaled w.r.t. the number of concurrent threads. Our algorithm retains polynomial complexity while gaining enough precision for proving correctness.

Can one enlarge this class of programs even more, for example, to the very general one given in 2.2.5? The extension lies in the arbitrary control flow structure, multiple locks and combinable critical sections in which several locks are syntactically acquired. We don't know the answer to this question, although we have already derived that there exists an exception set which is polynomially large:

Problem 3.6.1. Is class of lock programs presented in 2.2.5 verifiable in polynomial time by the CEGAR algorithm if the number of locks is constant?

3.7 Implementation AMTV

Now we briefly describe AMTV (Automatic Modular Thread Verifier), an implementation prototype of the CEGAR-TM algorithm.

The input to AMTV is the transition relation of the multithreaded program as in Def. 1.1.1 and the set of error states, given as a list of transitions for all $\rightarrow_i$ $(1 \leq i \leq n)$ and a list of erroneous program states. The output is "safe", equipped with an invariant in the union-of-products form, or "unsafe", equipped with a so-called error tree. The error tree is a sequence of iterates that starts with some of the initial states, each non-initial iterate is included in the set of successors of the previous iterate, and the last iterate contains some of the error states only.

We have experimented both with the locks class as well as with numerous small programs like the two-threaded Peterson's algorithm.

The implementation follows the above description of CEGAR-TM of the current chapter. We have experimented with the following variants.

- Notice that in the union-of-products representation each product is associated with a distinct shared state. Thus, given a product, it can be

immediately determined which transitions are applicable. We decided to store the transition relation not as a list of tuples, but as a search tree, in which transitions are sorted first by the source shared state. This variant had no significant influence upon the runtime of the mutex class (since there are two shared states only), but had a slight speedup (with a factor between 1 and 2) on examples with large shared state spaces.

- We have tried a so-called *eager* version of the main CEGAR loop, in which the extracted exception set was always applied at the very first iterate. Backtracking has two outcomes: a spurious trace occurs either due to approximation, or due to exception set containing a state leading to an error state. In the first case we enlarge the exception set as in the lazy CEGAR loop. In the second case we move states of the exception set that lead to an error state into the error set and restart the computation with a smaller exception set and a larger error set. There is no clear runtime gain or loss in general: the mutex class is analyzed slightly faster, the remaining programs slightly slower due to large representation of the growing error set. The speedup/slowdown factor was always between 1 and 2.

- We have tried both the eager and the lazy version of Extract using Fig. 3.5. In the lazy version, only one axis was used for extraction, while in the eager version all the axes on which A and Bad have disjoint projections are used for creation of exception sets which are then united. We have found that eager Extract is at least as good as the lazy Extract on all the examples.

Thus, currently AMTV uses the following defaults: transitions are stored in a tree sorted by the source shared state, the lazy variant of the outer CEGAR loop is used, and the eager variant of Extract is employed.

AMTV is written in OCAML, is licenced under GPL and is freely available from [10].

Open Problems

We have identified several nontrivial open problems so far. We restate a short version of these problems here.

- Problem 1.12.3. Prove that the diameter of a multithreaded program cannot be exponential in the number of threads if the number of shared states remains constant.
- Problem 2.2.11. Is there for each set of n-dimensional points in $(\mathbb{N}_d)^n$ a data structure of size $\text{poly}(d)^{\text{poly}(n)}$ such that checking whether an arbitrary product inside $(\mathbb{N}_d)^n$ is contained in this fixed set of points takes polynomial time in d and in n?
- Problem 3.6.1. Is the most general locks class of 2.2.5 polynomially verifiable for a fixed number of locks?

Conclusion

We have examined, reformulated and enhanced an approach for verification of concurrent programs. The first way of looking at this approach, based on Hoare-style manual verification, was discovered by Owicki and Gries [15] in 1976. Thread-modular model-checking, based on rely-guarantee reasoning, was introduced in 2003 by Flanagan and Qadeer [5]. Dependency-free analysis, later renamed into Cartesian abstraction, was described by Jones and Muchnick [8] in 1980. In 1985 Radhia Cousot [4] noticed that the proof method of Owicki and Gries is a form of abstract interpretation. Her work, first, neither contained a formal proof of this fact nor a connection to thread-modular reasoning, and, second, remained unknown. The Owicki-Gries and thread-modular methods were never connected to Cartesian abstraction by the majority of the community. We have rediscovered that connection, formally proving equivalence between all the three methods. The bridges between Cartesian abstraction, Owicki-Gries and thread-modular verification allow transfer of techniques known in one world into the other worlds.

The connection has been found, but all three techniques in their *basic* form are usable only for toy programs. Lack of precision is the biggest problem: most properties are just too complex for the simple techniques. The so-called standard solution of enlarging the shared state space to regain precision is not satisfactory: as of 2009, the author is not aware of any low-cost way of enlarging the shared state space - the costs incur either when a human provides manual verification support for a fast algorithm (e.g. invents auxiliary variables for Owicki-Gries) or when a machine does an exhaustive search (as in SPIN).

Nevertheless there have been two "standard" attacks (via enlarging shared state space) on the state-explosion problem for multithreaded programs recently. In [6] thread-modular verification is extended by making the guarantee stateful in a counterexample-guided manner. The technique seems interesting, but only results on unknown parameterized multithreaded integer programs are reported, and a working implementation is not available. In [2] auxiliary variables are automatically introduced by an eager CEGAR-loop. Despite encouraging results on the locks class and Peterson's algorithm, asymptotic runtime of their algorithm on examples is unknown, probably due to the unpredictable nature of BDDs.

We proceed differently from the very beginning. We know that proving safety of a finite multithreaded program is a PSPACE-complete problem. It turns out that the modification of the Cartesian abstraction proof by an exception set, which is described in this thesis, has complexity coNP or even P, depending on the representation of the exception set. And, in contrast to the "standard solution", the examined program remains the same - no auxiliary variables or

other forms of auxiliary shared state are added.

As soon as we have discovered exception sets, a great question appeared: how to find those nice and small exception sets, and how to do it automatically?

When a thread-modular proof fails, the reason for the failure is usually not pinpointable. In contrast, the reason for the imprecision (or failure) of Cartesian abstraction is precisely traceable - due to the nature of the fixpoint iteration. It turns out that we can just find the iterate in the Cartesian-abstraction-proof where too much precision is lost and introduce an exception set at this point. In case the new proof would fail once again, a new exception set would be found by looking at the new sequence of iterates.

This simple magic works. A practically interesting class of programs (the locks class) turned out to be verifiable even in polynomial time. The size of the required exception set for that class is polynomial, the time to find it by a CEGAR scheme is also polynomial. We should notice that the binary locks class is not amenable to the basic thread-modular or Owicki-Gries technique. Moreover, all programs which were amenable to thread-modular technique are polynomially verifiable by the new CEGAR technique too.

Happily, the theory gave rise to a usable implementation. The implementation is able to handle a hundred of threads on the locks class in polynomial time not only in theory, but also in practice.

Alexander Malkis, summer 2009. Revised: February 2010. Last updated: January 2024.

Glossary

Notation	Description	Page List
inductive invariant	A set of program states that contains the initial states and that is closed under the transition relation of the program.	17, 19, 23, 27, 28, 40, 42, 48, 53, 57
$\mathbb{N}^+$	The set of positive integers.	5, 7, 8, 15, 23, 58, 60, 69
$\mathbb{N}_0$	The set of nonnegative integers.	3, 7, 22, 23, 51, 52, 60, 69
$\mathbb{N}_n$ $(n \geq 1)$	$\mathbb{N}^+ \cap [1, n]$, the set of first n positive integers.	2–4, 6–8, 10–13, 15–19, 30–35, 37, 41–45, 61, 65, 69
power order	Given a poset $(X, \preceq)$ and $n \in \mathbb{N}_0$, the n^{th} *power order* is the product order on X^n.	6, 7
product order	Given posets $(X_i, \leq_i)$ and $n \in \mathbb{N}_0$, the *product order* on $L = \prod_{i=1}^n X_i$ is defined by $(a_i)_{i=1}^n \sqsubseteq (b_i)_{i=1}^n$ iff $\forall\, i \in \mathbb{N}_n : a_i \leq_i b_i$ $((a_i)_{i=1}^n, (b_i)_{i=1}^n \in L)$.	2, 3, 5, 6, 8, 9, 11–13, 17, 18, 22–24, 35
π_i	Given a product of sets $L = \prod_{i=1}^n X_i$ for $n \in \mathbb{N}^+$ and $i \in \mathbb{N}_n$, the projection onto the i^{th} component is the map $\pi_i \colon \mathfrak{P}(L) \to \mathfrak{P}(X_i)$, $S \mapsto \{l_i \mid l \in S\}$.	1, 5, 7, 15, 32, 35

Bibliography

[1] B. Blanchet. Introduction to abstract interpretation. Lecture script, `http://bblanche.gitlabpages.inria.fr/absint.pdf`, 2002.

[2] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 2007.

[3] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *FPCA*, pages 170–181, 1995.

[4] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles*. PhD thesis, Institut national polytechnique de Lorraine, 1985. Section 4.3.2.4.3, pages 4-118, 4-119, 4-120.

[5] C. Flanagan and S. Qadeer. Thread-modular model checking. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.

[6] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In W. Pugh and C. Chambers, editors, *PLDI*, pages 1–13. ACM, 2004.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[8] N. D. Jones and S. S. Muchnick. Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In *FOCS*, pages 185–190. IEEE, 1980.

[9] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266. IEEE, 1977.

[10] A. Malkis. AMTV: Automatic Modular Thread Verifier. `http://www.sec.in.tum.de/~malkis/Malkis-amtv.tar.7z`, implementation.

[11] A. Malkis and A. Podelski. Refinement with exceptions. Technical report, `http://software.imdea.org/~alexmalkis/refinementWithExceptions_techrep.pdf`, 2008.

[12] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification and Cartesian abstraction. Presentation at TV'06, 2006.

[13] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is Cartesian abstract interpretation. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2006.

[14] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1984.

[15] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[16] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.