

Thread-Modular Verification Is Cartesian Abstract Interpretation

Alexander Malkis¹, Andreas Podelski^{1,2}, and Andrey Rybalchenko^{1,3}

¹ Max-Planck Institut für Informatik, Saarbrücken

² Albert-Ludwigs-Universität Freiburg

³ EPFL IC IIF MTC, Lausanne

{malkis, podelski, rybal}@mpi-sb.mpg.de

Abstract. Verification of multithreaded programs is difficult. It requires reasoning about state spaces that grow exponentially in the number of concurrent threads. Successful verification techniques based on modular composition of over-approximations of thread behaviors have been designed for this task. These techniques have been traditionally described in assume-guarantee style, which does not admit reasoning about the abstraction properties of the involved compositional argument. Flanagan and Qadeer thread-modular algorithm is a characteristic representative of such techniques. In this paper, we investigate the formalization of this algorithm in the framework of abstract interpretation. We identify the abstraction that the algorithm implements; its definition involves Cartesian products of sets. Our result provides a basis for the systematic study of similar abstractions for dealing with the state explosion problem. As a first step in this direction, our result provides a characterization of a minimal increase in the precision of the Flanagan and Qadeer algorithm that leads to the loss of its polynomial complexity.

1 Introduction

Multithreaded software is everywhere. Verifying multithreaded software is an important and difficult task. The worst-case runtime of every existing verification algorithm is exponential in the number of threads. Due to inherent theoretical restrictions (see [9]) this is unlikely to change in the future.

However there are many successful algorithms and tools for different types of communication between components of a concurrent system. One can mention algorithms in SPIN (see [8]) and BLAST (see [7]) model-checkers. The main challenge for any verification algorithm is to reason modularly, avoiding explicit products of states and state spaces, which lead to combinatorial explosion. Such algorithms are called thread-modular.

Assume-guarantee reasoning offers a prominent approach to devise thread-modular algorithms. The behavior of each thread with respect to global variables is described by its guarantee. You can view this guarantee as a transition relation. For each thread, the model-checking procedure is applied to a parallel composition of the thread's transition relation and guarantees of other threads.

In this way the thread assumes that its environment sticks to the guarantees of other threads. Thus the guarantees of other threads represent the environment assumption. During model-checking of the parallel composition, the behavior of each thread is checked to stick to its own guarantee.

However, assume-guarantee reasoning doesn't provide an insight about the abstraction process involved. All that is known about precision loss during search in a thread's state space is that during discovering states of the thread, the behavior of all other threads is reduced to their action on global variables. Nothing was known about the loss of precision for the program as a whole. It was unclear whether or how is it possible to represent thread-modular reasoning in the standard framework of abstract interpretation.

As soon as the abstraction is identified, it provides additional insight into the algorithms. For instance, one could try to increase precision of the abstraction, to try to adapt the precision to a verification task, to optimize the algorithms, to combine with other abstractions or add refinement and create a counterexample-guided abstraction refinement loop. One could construct modifications of the abstraction, derive the corresponding algorithms and look at their runtime and precision.

We study the Flanagan-Qadeer algorithm for thread-modular verification (from now - FQ-algorithm). The distinguishing property of this algorithm lies in its low complexity. It is polynomial in the number of threads. The low complexity has its price: the algorithm is incomplete. The algorithm is also of the assume-guarantee type. Each computed guarantee is a set of pairs of valuations of unprimed and primed global variables (g, g') . While model-checking a thread, each time a thread state is discovered, we allow the global variables to be changed according to the guarantees of other threads. Each time the thread itself changes global variables from a discovered state, the corresponding pair of valuations of global variables before and after the step is added to the guarantee. Upon convergence, the environment assumptions are devised and the discovered states of each threads include the reachable thread states.

We would like to identify the abstraction used by the FQ-algorithm to be able to reason about the algorithm. In particular, we would like to know how far one can push the FQ-algorithm while still being polynomial in time and space.

In this paper we study the abstraction used in FQ-algorithm and identify the boundary. Our result is that FQ-algorithm implements Cartesian abstraction in special setting with threads (a so-called local Cartesian abstraction).

This insight allows us to find the "least modification" of the FQ-algorithm that leads to the loss of polynomial complexity. Thus, the identification of the abstraction provides the insight into the algorithm itself.

Local Cartesian abstraction formulation does not immediately provide a basis for complexity analysis, since the concrete domain (which is both the domain and the range of local Cartesian abstraction) admits exponentially long chains. Since local Cartesian abstraction is "the what" of FQ-algorithm, we obtain a polynomial time algorithm for reachability under local Cartesian abstraction.

Cartesian abstraction in program analysis is also known as “independent attribute method” (see [10]). There has been a lot of research since then, but to our best knowledge the method has not been applied to multithreaded programs yet. So our work is the first attempt to develop the theory of Cartesian abstraction for multithreaded programs.

Outline of the paper: First we define our program model.

Then we explain the algorithm. We provide a small example.

After that we define a concrete and an abstract domain and a corresponding Galois-connection that involve Cartesian products of sets. We state our first theorem saying how the output of the FQ-algorithm is expressible in the standard framework of abstract interpretation. We prove this theorem.

Then we define local Cartesian abstraction of multithreaded programs. We state our second theorem which says that the algorithm implements local Cartesian abstraction: the output of the algorithm represents local Cartesian abstraction of the program. We demonstrate the theorem on our example and prove it in general.

At last, we give a boundary of the abstraction, namely, an optimizing modification that works for many other verification methods but immediately breaks the polynomial-time border of the FQ-algorithm.

2 Preliminaries

2.1 Programs with Threads

We are interested in proving safety properties of multithreaded programs. Each safety property can be encoded as a reachability property. Simplifying, we consider programs consisting of only two threads communicating via shared variables.

A two-threaded *program* is given by a tuple

$$(\text{Glob}, \text{Loc}_1, \text{Loc}_2, \rightarrow_1, \rightarrow_2, \text{init})$$

where

- Loc_1 and Loc_2 contain valuations of local variables of the first and second threads, we call them the *local stores* of the first and second thread;
- Glob contains valuations of shared variables, we call it the *global store*;
- the elements of $\text{States} = \text{Glob} \times \text{Loc}_1 \times \text{Loc}_2$ are called *program states*, the elements of $Q_1 = \text{Glob} \times \text{Loc}_1$ and $Q_2 = \text{Glob} \times \text{Loc}_2$ are called *thread states*;
- the relation \rightarrow_1 (resp. \rightarrow_2) is a binary transition relation on the states of the first (resp. second) thread;
- $\text{init} \subseteq \text{States}$ is a set of initial states.

The program is equipped with the interleaving semantics. 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 either a step of the first or a step of the second thread. The following successor operator maps a set of program states to the set of their successors:

$$\begin{aligned} \text{post} : 2^{\text{States}} &\rightarrow 2^{\text{States}} \\ S &\mapsto \{(g', l'_1, l'_2) \mid \exists (g, l_1, l_2) \in S : \\ &\quad (g, l_1) \rightarrow_1 (g', l'_1) \text{ and } l_2 = l'_2 \\ &\quad \text{or } (g, l_2) \rightarrow_2 (g', l'_2) \text{ and } l_1 = l'_1\}. \end{aligned}$$

We are interested whether there is a computation of any length $k \geq 0$ that starts in an initial state and ends in a single user-given error state f , formally:

$$\exists k \geq 0 : f \in \text{post}^k(\text{init}).$$

2.2 Flanagan-Qadeer Algorithm

The FQ-algorithm from [6] tests whether a given bad state f is reachable from an initial state. The test says “no” or “don’t know”.

The algorithm computes sets $\mathcal{R}_i \subseteq \text{Glob} \times \text{Loc}_i$ and $\mathcal{G}_i \subseteq \text{Glob} \times \text{Glob}$ ($i = 1, 2$) defined by the least fixed point of the following inference rules:

$$\begin{aligned} \text{INIT} &\frac{}{\text{init}_i \in \mathcal{R}_i} & \text{STEP} &\frac{(g, l) \in \mathcal{R}_i \quad (g, l) \rightarrow_i (g', l')}{(g', l') \in \mathcal{R}_i \quad (g, g') \in \mathcal{G}_i} \\ \text{ENV} &\frac{(g, l) \in \mathcal{R}_i \quad (g, g') \in \mathcal{G}_j \quad i \neq j}{(g', l) \in \mathcal{R}_i} \end{aligned}$$

Here, $\text{init}_1 = \{(g, l_1) \mid (g, l_1, _) \in \text{init}\}$, similarly $\text{init}_2 = \{(g, l_2) \mid (g, _, l_2) \in \text{init}\}$. (The underscore means “anything”, i.e. an existentially quantified variable. The quantification is innermost, so in a formula, two underscores at different places denote different existentially quantified variables.) If $f \in \{(g, l_1, l_2) \mid (g, l_1) \in \mathcal{R}_1 \text{ and } (g, l_2) \in \mathcal{R}_2\}$, the algorithm says “don’t know”, otherwise it says “no”.

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. After the fixed point is reached, the set \mathcal{R}_1 (resp. \mathcal{R}_2) contains those states of the first (resp. second) thread that the algorithm discovers. The discovered thread states contain those thread states that occur in computations.

3 Represented Program States

The inference rules of the FQ-algorithm define the sets $\mathcal{R}_1, \mathcal{R}_2$ of “discovered” thread states. These sets represent those program states, whose globals and locals of the first thread are in \mathcal{R}_1 and globals and locals of the second thread are in \mathcal{R}_2 , namely $\{(g, l_1, l_2) \mid (g, l_1) \in \mathcal{R}_1 \text{ and } (g, l_2) \in \mathcal{R}_2\}$.

Here is a small example. The program below has one global variable g that can take values 0 or 1, the first (resp. second) thread has a single local variable pc_1 (resp. pc_2), representing the program counter.

$$\begin{array}{l} \text{Initially } g = 0 \\ A : g := 0; \qquad C : g := 1; \\ B : \qquad \qquad \qquad D : \end{array}$$

The algorithm discovers the following thread states:

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

where (x, Y) is a shorthand for the the pair of two maps $([g \mapsto x], [pc_i \mapsto Y])$. These two sets represent the set of program states

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

where (x, Y, Z) means the triple of maps $([g \mapsto x], [pc_1 \mapsto Y], [pc_2 \mapsto Z])$.

4 Cartesian Abstract Interpretation

In order to characterize the FQ-algorithm in the abstract interpretation framework, we first need a concrete domain, an abstract domain and a Galois connection between them:

$$\begin{aligned} D &= 2^{\text{States}} \text{ is the set underlying the concrete lattice,} \\ D^\# &= 2^{Q_1} \times 2^{Q_2} \text{ is the set underlying the abstract lattice,} \\ \alpha_{\text{cart}} : D &\rightarrow D^\#, \quad S \mapsto (T_1, T_2) \text{ where } T_1 = \{(g, l) \mid (g, l, _) \in S\} \\ & \qquad \qquad \qquad T_2 = \{(g, l) \mid (g, _, l) \in S\}, \\ \gamma_{\text{cart}} : D^\# &\rightarrow D, \quad (T_1, T_2) \mapsto \{(g, l_1, l_2) \mid (g, l_1) \in T_1 \text{ and } (g, l_2) \in T_2\}. \end{aligned}$$

The ordering on the concrete lattice D is inclusion, the least upper bound is the union \cup , the greatest lower bound is the intersection \cap .

The ordering on the abstract lattice $D^\#$ is the product ordering, i.e. $(T_1, T_2) \sqsubseteq (T'_1, T'_2)$ if and only if $T_1 \subseteq T'_1$ and $T_2 \subseteq T'_2$. The least upper bound \sqcup is componentwise union, the greatest lower bound \sqcap is componentwise intersection.

Remark that the image of the abstraction map α_{cart} is always contained in

$$D^{\#\dagger} = \{(T_1, T_2) \in D^\# \mid \forall g \in \text{Glob} : (g, _) \in T_1 \Leftrightarrow (g, _) \in T_2\}.$$

Now we show that for the finite-state case the maximal chain length of the abstract domain is in general smaller than that of the concrete domain.

Proposition 1. *Let $\text{Glob}, \text{Loc}_1, \text{Loc}_2$ be finite. Let $G := |\text{Glob}| \geq 1$ be the cardinality of the global store and $L_1 := |\text{Loc}_1|$, $L_2 := |\text{Loc}_2|$ be the cardinalities of the local stores, both at least 2 and $l := \min\{L_1, L_2\}$. Then the maximal chain length of abstract domain is smaller and also asymptotically smaller than the maximal chain length of the concrete domain. Formally:*

- a) (maximal chain length of $D^\#$) \leq (maximal chain length of D);
- b) $\lim_{l \rightarrow \infty} \frac{\text{maximal chain length of } D^\#}{\text{maximal chain length of } D} = 0$.

Proof. a) Consider any longest chain in D and any adjacent elements $A \subset B$ in the chain. Then $A \dot{\cup} \{_\} = B$ (otherwise the chain could be lengthened). So if \emptyset is the 0th element of the chain, then its i th element from the bottom has size i . The maximal element is States, so the chain has $1 + |\text{States}| = 1 + GL_1L_2$ elements.

Consider any longest chain in $D^\#$ and any two adjacent elements $(A_1, A_2) \sqsubset (B_1, B_2)$ in the chain. Then either $A_1 = B_1$ and $A_2 \dot{\cup} \{_\} = B_2$ or $A_2 = B_2$ and $A_1 \dot{\cup} \{_\} = B_1$ (otherwise the chain could be lengthened). So one can construct the chain by starting with (\emptyset, \emptyset) and adding elements one by one in some order to the first or to the second component. The number of such additions is bounded by the maximal sizes of the components $|Q_1|$ and $|Q_2|$. Totally $|Q_1| + |Q_2|$ additions can be performed, so the chain has $1 + |Q_1| + |Q_2| = 1 + GL_1 + GL_2$ elements.

b)

$$\lim_{l \rightarrow \infty} \frac{\text{maximal chain length of } D^\#}{\text{maximal chain length of } D} = \lim_{l \rightarrow \infty} \frac{1 + G(L_1 + L_2)}{1 + GL_1L_2} =$$

$$\underbrace{\lim_{l \rightarrow \infty} \frac{1}{1 + GL_1L_2}}_0 + \lim_{l \rightarrow \infty} \frac{G(L_1 + L_2)}{1 + GL_1L_2} = \lim_{l \rightarrow \infty} \frac{1}{\frac{1}{GL_1} + L_2} + \lim_{l \rightarrow \infty} \frac{1}{\frac{1}{GL_2} + L_1} = 0.$$

□

Two remarks should be made. First, if only one local store grows but the other remains constant-size, then the quotient $\frac{\text{maximal chain length of } D^\#}{\text{maximal chain length of } D}$ approaches some small positive value between 0 and 1. In case the number of threads is not two, but variable (say, n), we get similar asymptotic results for $n \rightarrow \infty$.

From now on, we sometimes omit the parentheses around the argument of a map, writing, e.g. fx for $f(x)$.

A pair of maps (α, γ) with $\alpha : D \rightarrow D^\#$ and $\gamma : D^\# \rightarrow D$ is called a *Galois connection* if for all $S \in D, T \in D^\#$ we have: $\alpha S \subseteq T$ iff $S \subseteq \gamma T$.

Proposition 2. *The pair of maps $(\alpha_{\text{cart}}, \gamma_{\text{cart}})$ is a Galois connection, formally:*

$$\forall S \in D, (T_1, T_2) \in D^\# : \quad \alpha_{\text{cart}} S \subseteq (T_1, T_2) \text{ iff } S \subseteq \gamma_{\text{cart}}(T_1, T_2).$$

Proof. “ \Rightarrow ”: Let $(g, l_1, l_2) \in S$. Let $(T'_1, T'_2) = \alpha_{\text{cart}} S$. Then by definition of α_{cart} we have $(g, l_1) \in T'_1 \subseteq T_1$ and $(g, l_2) \in T'_2 \subseteq T_2$. So $(g, l_1, l_2) \in \gamma_{\text{cart}}(T_1, T_2)$ by definition of γ_{cart} .

“ \Leftarrow ”: Let $(T'_1, T'_2) = \alpha_{\text{cart}} S$. Let $(g, l_1) \in T'_1$. By definition of α_{cart} there is an l_2 with $(g, l_1, l_2) \in S \subseteq \gamma_{\text{cart}}(T_1, T_2)$. By definition of γ_{cart} we have $(g, l_1) \in T_1$. So $T'_1 \subseteq T_1$. Analogously we get $T'_2 \subseteq T_2$. □

5 Flanagan-Qadeer Algorithm Implements Cartesian Abstract Fixpoint Checking

Given a Galois connection (α, γ) between an abstract and a concrete domain, the abstraction of the program is defined as the least fixed point of $\lambda T. \alpha(\text{init} \cup \text{post}\gamma T)$ (see e.g. [5]). Recall that the FQ-algorithm computes \mathcal{R}_1 and \mathcal{R}_2 , the sets of “discovered” states of the first and second thread.

Theorem 3. [*Thread-Modular Model Checking is Cartesian Abstract Interpretation*]

The output of the FQ-algorithm is the least fixed point of the abstract fixpoint checking operator with the abstraction map α_{cart} and concretization map γ_{cart} . Formally:

$$(\mathcal{R}_1, \mathcal{R}_2) = \text{lfp } \lambda T. \alpha_{\text{cart}}(\text{init} \cup \text{post}\gamma_{\text{cart}} T).$$

It is not clear why this is so and how the assumptions are connected. For our tiny example, the right hand of the above equation (i.e. the least fixed point) 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 algorithm. We prove that the left and right hand side always coincide in the next section.

6 Proof

First we transform the inference rules of the FQ-algorithm by getting rid of the sets \mathcal{G}_1 and \mathcal{G}_2 . We get an equivalent system of inference rules

$$\begin{array}{l} \text{INIT}'_1 \frac{}{\text{init}_1 \in \mathcal{R}_1} \qquad \text{STEP}'_1 \frac{(g, l) \in \mathcal{R}_1 \quad (g, l) \rightarrow_1 (g', l')}{(g', l') \in \mathcal{R}_1} \\ \text{ENV}'_1 \frac{(g, l) \in \mathcal{R}_1 \quad (g, l_2) \in \mathcal{R}_2 \quad (g, l_2) \rightarrow_2 (g', _)}{(g', l) \in \mathcal{R}_1} \end{array}$$

The rules INIT'_2 , STEP'_2 and ENV'_2 are accordingly to INIT'_1 , STEP'_1 and ENV'_1 where the indices 1 and 2 are exchanged. Remark that init_1 and init_2 contain thread states with the same global parts. Also remark that whenever $(g, l) \in \mathcal{R}_1$ and $(g, l) \rightarrow_1 (g', _)$ and there is some thread state $(g, _)$ in \mathcal{R}_2 , then both rules STEP'_1 and ENV'_2 apply, giving two thread states for \mathcal{R}_1 and \mathcal{R}_2 with the same global part g' . Similarly, whenever $(g, l) \in \mathcal{R}_2$ and $(g, l) \rightarrow_2 (g', _)$ and there is some thread state $(g, _)$ in \mathcal{R}_1 , then both rules STEP'_2 and ENV'_1 apply, giving two thread states for \mathcal{R}_2 and \mathcal{R}_1 with the same global part g' . By induction follows that whenever there is a thread state in \mathcal{R}_i with some global part g , there is a thread state in \mathcal{R}_j with the same global part g ($i \neq j$).

This means that we can replace the STEP' and ENV' rules by one rule. The following system of inference rules is equivalent to the system above.

$$\begin{array}{l}
 \text{INIT}'_1 \frac{}{\text{init}_1 \in \mathcal{R}_1} \qquad \qquad \qquad \text{INIT}'_2 \frac{}{\text{init}_2 \in \mathcal{R}_2} \\
 \text{POST}'_1 \frac{(g, l_1) \in \mathcal{R}_1 \quad (g, l_2) \in \mathcal{R}_2 \quad (g, l_2) \rightarrow_2 (g', l'_2)}{(g', l_1) \in \mathcal{R}_1 \quad (g', l'_2) \in \mathcal{R}_2} \\
 \text{POST}'_2 \frac{(g, l_2) \in \mathcal{R}_2 \quad (g, l_1) \in \mathcal{R}_1 \quad (g, l_1) \rightarrow_1 (g', l'_1)}{(g', l_2) \in \mathcal{R}_2 \quad (g', l'_1) \in \mathcal{R}_1}
 \end{array}$$

Each $\text{POST}^\#$ rule takes two sets (called \mathcal{R}_1 and \mathcal{R}_2 above) and gives new elements for the first and new elements for the second set. All possible applications of the $\text{POST}^\#$ rules on a fixed pair of sets (T_1, T_2) can be expressed as computing

$$\begin{aligned}
 p^\#(T_1, T_2) = \{ & ((g', l'_1), (g', l'_2)) \mid \exists ((g, l_1), (g, l_2)) \in T_1 \times T_2 : \\
 & (g, l_1) \rightarrow_1 (g', l'_1) \text{ and } l_2 = l'_2 \\
 & \text{or } (g, l_2) \rightarrow_2 (g', l'_2) \text{ and } l_1 = l'_1 \},
 \end{aligned}$$

the new elements being the first and second projection of the result. Thus, applying the $\text{POST}^\#$ rules corresponds to applying the map $\text{post}^\# : D^\# \rightarrow D^\#$,

$$(T_1, T_2) \mapsto (\pi_1 p^\#(T_1, T_2), \pi_2 p^\#(T_1, T_2)),$$

where π_i is the projection on the i th component ($i = 1, 2$). Notice: $(\text{init}_1, \text{init}_2) = \alpha_{\text{cart}} \text{init}$. Then the pair of computed sets $(\mathcal{R}_1, \mathcal{R}_2)$ is the least fixed point of

$$\lambda T. \alpha_{\text{cart}} \text{init} \sqcup \text{post}^\# T.$$

The abstract successor map $\text{post}^\#$ can be expressed in terms of post and the abstraction/concretization maps:

Proposition 4. *For any $T \in D^\#$ holds:*

$$\text{post}^\# T = \alpha_{\text{cart}} \text{post} \gamma_{\text{cart}} T.$$

Proof. Let $(T_1, T_2) := T$.

“ \sqsubseteq ”:

Let $(g', l'_1) \in \pi_1 p^\# T$. Then there is an l'_2 so that the pair $((g', l'_1), (g', l'_2)) \in p^\# T$. Then there are g, l_1, l_2 with $(g, l_1) \in T_1, (g, l_2) \in T_2$ and

$$\begin{aligned}
 & (g, l_1) \rightarrow_1 (g', l'_1) \text{ and } l_2 = l'_2 \\
 & \text{or } (g, l_2) \rightarrow_2 (g', l'_2) \text{ and } l_1 = l'_1.
 \end{aligned}$$

Then $(g, l_1, l_2) \in \gamma_{\text{cart}}(T_1, T_2) = \gamma_{\text{cart}} T$ by definition of γ_{cart} . So $(g', l'_1, l'_2) \in \text{post}_{\gamma_{\text{cart}}} T$ by definition of the successor map post and thus (g', l'_1) is in the first component of $\alpha_{\text{cart}} \text{post}_{\gamma_{\text{cart}}} T$. So $\pi_1 p^\# T$ is contained in the first component of $\alpha_{\text{cart}} \text{post}_{\gamma_{\text{cart}}} T$. That $\pi_2 p^\# T$ is contained in the second component of $\alpha_{\text{cart}} \text{post}_{\gamma_{\text{cart}}} T$ can be proven analogously.

“ \supseteq ”:

Let (g', l'_1) be in the first component of $\alpha_{\text{cart}} \text{post}_{\gamma_{\text{cart}}} T$. Then there is an l'_2 with $(g', l'_1, l'_2) \in \text{post}_{\gamma_{\text{cart}}} T$. So there are g, l_1, l_2 with $(g, l_1, l_2) \in \gamma_{\text{cart}} T$ and

$$\begin{aligned}
 & (g, l_1) \rightarrow_1 (g', l'_1) \text{ and } l_2 = l'_2 \\
 & \text{or } (g, l_2) \rightarrow_2 (g', l'_2) \text{ and } l_1 = l'_1.
 \end{aligned}$$

From $(g, l_1, l_2) \in \gamma_{\text{cart}}T$ we know that $(g, l_1) \in T_1$ and $(g, l_2) \in T_2$. By definition of $p^\#$ we have $((g', l'_1), (g', l'_2)) \in p^\#(T_1, T_2)$. So $(g', l'_1) \in \pi_1 p^\#T$. We have shown that the first component of $\alpha_{\text{cart}}\text{post}\gamma_{\text{cart}}T$ is included in $\pi_1 p^\#T$. Analogously one can show that the second component of $\alpha_{\text{cart}}\text{post}\gamma_{\text{cart}}T$ is included in $\pi_2 p^\#T$. \square

So the algorithm computes the least fixed point of

$$\lambda T . \alpha_{\text{cart}}\text{init} \sqcup \alpha_{\text{cart}}\text{post}\gamma_{\text{cart}}T = \lambda T . \alpha_{\text{cart}}(\text{init} \cup \text{post}\gamma_{\text{cart}}T).$$

7 Local Cartesian Abstraction

Up to now we identified the FQ-algorithm as abstract fixpoint checking on an abstract domain. However, it turns out that the output of the FQ-algorithm can also be characterized by a very simple abstraction of multithreaded programs which is defined on the concrete domain. Now we define this abstraction.

Recall that the *Cartesian abstraction of a set of pairs* is the smallest Cartesian product containing this subset. We define it formally as

$$\begin{aligned} \mathcal{C}^\# : 2^{Q_1 \times Q_2} &\rightarrow 2^{Q_1 \times Q_2}, \\ P &\mapsto \{(s_1, s_2) \mid (s_1, _) \in P \text{ and } (_, s_2) \in P\}, \end{aligned}$$

We have $\mathcal{C}^\#P = \pi_1 P \times \pi_2 P$. An analog of Cartesian abstraction on the concrete domain is

$$\begin{aligned} \mathcal{C} : D &\rightarrow D \\ S &\mapsto \{(g, l_1, l_2) \mid (g, l_1, _) \in S \text{ and } (g, _, l_2) \in S\}. \end{aligned}$$

We call this map *local Cartesian abstraction* of a set of program states since it simplifies to the Cartesian abstraction of a set of pairs if Glob is a singleton.

It turns out that local Cartesian abstraction is representable in the abstract interpretation framework.

Proposition 5. *Local Cartesian abstraction is overapproximation with the abstraction map α_{cart} and the concretization map γ_{cart} . Formally:*

$$\mathcal{C} = \gamma_{\text{cart}}\alpha_{\text{cart}}$$

Proof. Let $S \subseteq \text{States}$. We show that $\mathcal{C}S = \gamma_{\text{cart}}\alpha_{\text{cart}}S$. Let $(T_1, T_2) = \alpha_{\text{cart}}S$. We have

$$\begin{aligned} (g, l_1, l_2) \in \mathcal{C}S &\stackrel{\text{def. of } \mathcal{C}}{\iff} (g, l_1, _) \in S \text{ and } (g, _, l_2) \in S \stackrel{\text{def. of } \alpha_{\text{cart}}}{\iff} (g, l_1) \in T_1 \\ \text{and } (g, l_2) \in T_2 &\stackrel{\text{definition of } \gamma_{\text{cart}}}{\iff} (g, l_1, l_2) \in \gamma_{\text{cart}}(T_1, T_2) = \gamma_{\text{cart}}\alpha_{\text{cart}}S. \quad \square \end{aligned}$$

8 Thread-Modular Model-Checking as Local Cartesian Abstraction

Given an abstraction map α and a concretization map γ between an abstract and a concrete domain, we can alternatively perform the abstract fixed point

checking in the concrete domain. Then the least fixed point of $\lambda S. \gamma \alpha(\text{init} \cup \text{post} S)$ is computed.

For our special Galois connection $(\alpha_{\text{cart}}, \gamma_{\text{cart}})$ we can discover the states of the program by iterative successor computation and at each step overapproximate by local Cartesian abstraction. Naively implemented, this algorithm would require exponential time (in number of threads, if it is not constant). It turns out that the FQ-algorithm solves the same problem in polynomial time.

Theorem 6 (Thread-Modular Model-Checking as Local Cartesian abstraction). *The concretization of the output of the FQ-algorithm is equal to the result of abstract fixpoint checking with local Cartesian abstraction. Formally:*

$$\gamma(\mathcal{R}_1, \mathcal{R}_2) = \text{lfp } \lambda S. \mathcal{C}(\text{init} \cup \text{post} S). \quad (1)$$

For our tiny example, let us compute the least fixed point of $\lambda S. \mathcal{C}(\text{init} \cup \text{post} S)$ by definition. The corresponding chain is

$$\begin{aligned} & \{(0, A, C)\} \sqsubseteq \{(0, A, C), (0, B, C), (1, A, D)\} \sqsubseteq \\ & \sqsubseteq \{(0, A, C), (0, B, C), (1, A, D), (1, B, D), (0, B, D), (0, A, D)\} \quad (\text{fixed point}), \end{aligned}$$

the last term being the right hand side of (1). The left and the right hand side coincide in this example. We prove that they always coincide in the next section.

9 Proof

9.1 Preparations

Before we start proving the theorem, let's prove a basic fact about the abstract fixpoint checking.

Let D be a complete lattice with ordering \subseteq , bottom element \emptyset , join \cup , meet \cap (concrete lattice). Further, let $D^\#$ be a complete lattice with ordering \sqsubseteq , bottom element \perp , join \sqcup and meet \sqcap (abstract lattice). Let a pair of maps $\alpha : D \rightarrow D^\#$ and $\gamma : D^\# \rightarrow D$ be a Galois connection between the concrete and abstract lattices. Let $F : D \rightarrow D$ be any monotone map and $\text{init} \in D$ any concrete element. Further, we call $\rho := \gamma \alpha : D \rightarrow D$ the *overapproximation* operator.

One way to perform abstract fixpoint checking is to compute the least fixed point of $G^\# = \lambda T. \alpha(\text{init} \cup F \gamma T)$ in the abstract lattice. The other way is to compute the least fixed point of $G = \lambda S. \gamma \alpha(\text{init} \cup F S)$ in the concrete lattice.

One would expect that these two fixed points are the same up to abstraction/concretization. Now we show that this is indeed the case if we assume the following

Hypothesis. The concretization map γ is semi-continuous, i.e. for all ascending chains $X \subseteq D^\#$ we have $\gamma(\sqcup X) = \cup \gamma X$.

This hypothesis is especially satisfied for a continuous γ , i.e. when for all chains $X \subseteq D^\#$ we have $\gamma(\sqcup X) = \cup \gamma X$.

Let μ be any ordinal whose cardinality (the cardinality of the class of ordinals smaller than μ) is greater than the cardinalities of D and $D^\#$. Let's define two sequences with indices from μ :

$$\begin{array}{lll}
T^0 = \alpha \text{init} & S^0 = \rho \text{init} & \text{for } k = 0, \\
T^{k+1} = G^\# T^k & S^{k+1} = G S^k & \text{for successor ordinals } k+1 \in \mu, \\
T^k = \bigsqcup_{k' < k} T^{k'} & S^k = \bigcup_{k' < k} S^{k'} & \text{for limit ordinals } k \in \mu.
\end{array}$$

From [3] Cor 3.3 we know that the sequences $(S^k)_k$, $(T^k)_k$ are stationary increasing chains and the limits are the least fixed points over αinit and ρinit , respectively. One can show that if we start the sequences $(T^k)_k$ and $(S^k)_k$ with the bottom elements \perp and \emptyset (instead of αinit and ρinit), the limits would be the same, respectively. So the limits are the least fixed points (over \perp and \emptyset). The hypothesis immediately implies that for each limit ordinal $k \in \mu$ holds

$$\gamma \bigsqcup_{k' < k} T^{k'} = \bigcup_{k' < k} \gamma T^{k'} \quad \text{and} \quad \gamma \bigsqcup_{k' < k} \alpha S^{k'} = \bigcup_{k' < k} \gamma \alpha S^{k'}. \quad (2)$$

We need some basic facts about Galois connections and about the overapproximation operator (see e.g. [2], [4]). From the definition of Galois connection one can prove that the abstraction and concretization maps α and γ are monotone. Further overapproximation map ρ is idempotent. Further, overapproximating and then abstracting is the same as abstracting: $\alpha \rho = \alpha$.

First we show that overapproximating S^k doesn't change it.

Proposition 7. *Each S^k is invariant under overapproximation. Formally:*

$$\forall k \in \mu : \rho S^k = S^k.$$

Proof. We use transfinite induction.

For $k = 0$, we have $\rho S^0 = \rho \rho \text{init} \stackrel{\rho \text{ idempotent}}{=} \rho \text{init} = S^0$.

For a successor ordinal $k + 1$, we have $\rho S^{k+1} = \rho \rho (\text{init} \cup F S^k) \stackrel{\rho \text{ idempotent}}{=} \rho (\text{init} \cup F S^k) = S^{k+1}$.

If k is a limit ordinal, $\rho S^k = \gamma \alpha \bigcup_{k' < k} S^{k'} \stackrel{\alpha \text{ complete join morphism}}{=} \gamma \bigsqcup_{k' < k} \alpha S^{k'}$
 $\stackrel{\text{formula(2)}}{=} \bigcup_{k' < k} \gamma \alpha S^{k'} \stackrel{\rho = \gamma \alpha}{=} \bigcup_{k' < k} \rho S^{k'} \stackrel{\text{induction assumption}}{=} \bigcup_{k' < k} S^{k'} = S^k. \quad \square$

Proposition 8. *Each T^k is the abstraction of S^k . Formally:*

$$\forall k \in \mu : T^k = \alpha S^k.$$

Proof. Transfinite induction.

For $k = 0$ we have $T^0 = \alpha \text{init} \stackrel{\alpha = \alpha \rho}{=} \alpha \rho \text{init} = \alpha S^0$.

For a successor ordinal $k + 1$ we have $T^{k+1} = \alpha (\text{init} \cup F \gamma T^k) \stackrel{\text{induction assumption}}{=} \alpha (\text{init} \cup F \gamma \alpha S^k) \stackrel{\gamma \alpha = \rho}{=} \alpha (\text{init} \cup F \rho S^k) \stackrel{\rho S^k = S^k}{=} \alpha (\text{init} \cup F S^k) \stackrel{\alpha = \alpha \rho}{=} \alpha \rho (\text{init} \cup F S^k) = \alpha S^{k+1}$.

For a limit ordinal k holds $T^k = \bigsqcup_{k' < k} T^{k'} \stackrel{\text{induction assumption}}{=} \bigsqcup_{k' < k} \alpha S^{k'}$
 $\stackrel{\alpha \text{ complete join morphism}}{=} \alpha \bigcup_{k' < k} S^{k'} = \alpha S^k. \quad \square$

Proposition 9. *Each S^k is the concretization of T^k . Formally:*

$$\forall k \in \mu : \quad \gamma T^k = S^k.$$

Proof. Transfinite induction.

For $k = 0$ we have $\gamma T^0 = \gamma \alpha \text{init} = \rho \text{init} = S^0$.

For a successor ordinal $k + 1$ we have $\gamma T^{k+1} = \gamma \alpha(\text{init} \cup F \gamma T^k) = \rho(\text{init} \cup F \gamma T^k) \stackrel{\text{induction assumption}}{=} \rho(\text{init} \cup F S^k) = S^{k+1}$.

If k is a limit ordinal, $\gamma T^k \stackrel{\text{induction assumption}}{=} \gamma \bigsqcup_{k' < k} T^{k'} \stackrel{\text{formula(2)}}{=} \bigsqcup_{k' < k} \gamma T^{k'}$
 $\stackrel{\text{induction assumption}}{=} \bigsqcup_{k' < k} S^{k'} = S^k. \quad \square$

Let $\lambda \in \mu$ be any ordinal at which both sequences are stationary, i.e. $S^\lambda = S^{\lambda+1}$ and $T^\lambda = T^{\lambda+1}$. Then the least fixed point of G is S^λ and the least fixed point of $G^\#$ is T^λ . Propositions 8 and 9 imply the following

Theorem 10. *Let the concretization map be semi-continuous. Then the least fixed points of G and $G^\#$ coincide up to abstraction and concretization:*

$$\gamma \text{lfp } G^\# = \text{lfp } G \quad \text{and} \quad \text{lfp } G^\# = \alpha \text{lfp } G.$$

9.2 Applying the Theory

We now show that for our Galois connection $(\alpha_{\text{cart}}, \gamma_{\text{cart}})$ the hypothesis holds.

Proposition 11. *γ_{cart} is continuous, i.e. for all chains $X \subseteq D^\#$ holds:*

$$\gamma_{\text{cart}}(\bigsqcup X) = \cup \gamma_{\text{cart}} X.$$

Proof. “ \subseteq ”. Let $(g, l_1, l_2) \in \gamma_{\text{cart}}(\bigsqcup X)$. Then (g, l_1) (resp. (g, l_2)) is in the first (resp. second) component of $\bigsqcup X$. Then there are (T_1, T_2) and (T'_1, T'_2) in X with $(g, l_1) \in T_1$ and $(g, l_2) \in T'_2$. Since X is a chain, we have either $(T_1, T_2) \supseteq (T'_1, T'_2)$ or $(T_1, T_2) \sqsubseteq (T'_1, T'_2)$. Without loss of generality let $(T_1, T_2) \supseteq (T'_1, T'_2)$. Then $(g, l_2) \in T_2$, so $(g, l_1, l_2) \in \gamma_{\text{cart}}(T_1, T_2) \subseteq \cup \gamma_{\text{cart}} X$.

“ \supseteq ” holds by monotonicity of γ_{cart} and definition of the least upper bound. \square

The map $\text{post} : D \rightarrow D$ is monotone. Proposition 5 and Theorems 3 and 10 imply

$$\gamma(\mathcal{R}_1, \mathcal{R}_2) = \text{lfp } \lambda S. \mathcal{C}(\text{init} \cup \text{post} S).$$

10 Boundary of the Flanagan-Qadeer Algorithm

Now we try to push the FQ-algorithm to increase precision without losing polynomial complexity and show where this fails. For speaking about runtime, let’s assume that all the domains are finite. The definitions of the multithreaded program, of the concrete and the abstract domain, of abstraction/concretization maps and the corresponding theorems extend to n threads in a natural way.

A usual way to gain more precision is to abstract not all states, but only the recently discovered states:

$$T^0 = \alpha_{\text{cart}} \text{init} \quad \text{and} \quad T^{i+1} = \text{post}^\# T^i \quad (i \geq 0).$$

The sequence stops for $k \geq 0$ with $\gamma_{\text{cart}} T^{k+1} \subseteq \cup_{i=0}^k \gamma_{\text{cart}} T^i$. Then $X := \cup_{i=0}^k \gamma_{\text{cart}} T^i$ is an inductive invariant, i.e. $\text{init} \subseteq X$ and $\text{post}X \subseteq X$.

We can implement this iteration in the abstract domain $D^\#$ by the following inference rule:

$$\text{POST}_{ij}^\# \frac{(g, l_i) \in \mathcal{R}_i \quad (g, l_j) \in \mathcal{R}_j \quad (g, l_j) \rightarrow_j (g', l'_j)}{(g', l_i) \in \mathcal{R}'_i \quad (g', l'_j) \in \mathcal{R}'_j} \quad i \neq j.$$

Except for the primed versions \mathcal{R}'_i and \mathcal{R}'_j in the conclusion of the rule, this is the same rule that is used in the reformulation of the FQ-algorithm. As before we have $(\mathcal{R}'_1, \dots, \mathcal{R}'_n) = \text{post}^\#(\mathcal{R}_1, \dots, \mathcal{R}_n)$. So each steps of the new iteration scheme is polynomial. But it turns out that number of steps can be exponential:

Theorem 12. *Frontier search with Cartesian abstraction has exponential worst-case runtime in the number of threads.*

Proof. It suffices to present a family of multithreaded programs so that:

1. the n th program in the family has n threads;
2. the sizes of the global store and local stores are polynomial in n ;
3. each program of the family has exactly one run of exponential length in n .

Assume such a program with a single initial state is given. If for some $i \geq 0$ the components of the tuple T^i contain at most one element each, then $\gamma_{\text{cart}} T^i$ contains at most one element, and hence $\text{post} \gamma_{\text{cart}} T^i$ is a singleton or empty, so $T^{i+1} = \alpha_{\text{cart}} \text{post} \gamma_{\text{cart}} T^i$ is a tuple of singletons or empty sets. Since $\gamma_{\text{cart}} T^i$ contains $\text{post}^i(\text{init})$, we inductively follow that $\gamma_{\text{cart}} T^i = \text{post}^i(\text{init})$ for all $i \geq 0$, i.e. no approximation happens. Especially the sequence $(T^i)_{0 \leq i \leq k}$ is exponentially long.

Now we give a family of programs satisfying the conditions above.

Example 13. [Binary Counter] The statements in brackets $\langle \rangle$ are atomic.

Global boolean variable with initial value:

$t = 1$ (takes values from $\{0, \dots, n\}$)

Thread 1:

0: wait until $t = 1$;

1: $\langle t := 2; \text{ goto } 0; \rangle$

Thread i ($1 < i < n$):

0: $\langle \text{wait until } t = i; \quad t := 1; \rangle$

1: $\langle \text{wait until } t = i; \quad t := i + 1; \quad \text{goto } 0; \rangle$

Thread n :

0: $\langle \text{wait until } t = n; \quad t := 1; \rangle$

1: $\langle \text{wait until } t = n; \quad t := 0; \quad \text{goto } 0; \rangle$

The program implements a binary counter with the school addition method. The local store 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 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	1 0
pc_2	0 0	0 1	1 1	1 0	0 0	0 1	1 1	1 0
pc_3	0 0	0 0	0 0	0 1	1 1	1 1	1 1	1 0

Let us look at the columns marked by the star (*), i.e where carry is above the 0th position. The values of the program counters (pc_3, pc_2, pc_1) evolve like a binary counter. □

So frontier search breaks the polynomial time border of Cartesian Abstraction.

Another interesting property of the binary counter is that the set of reachable states is so big that the output of the FQ-algorithm is exact:

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

Namely, $\gamma_{\text{cart}}(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3)$ is exactly the set of reachable states. So no more precision can be regained.

The binary counter has a property that the size of the global store grows linearly with n and the transition graph has exponential diameter. Can one get exponential diameter with a sublinear or even constant global store size? We pose the following open

Problem 14. Prove or give a counterexample. There is no family $(P_n)_{n \geq 1}$ of multithreaded programs so that

1. the n th program P_n consists of n threads;
2. the global and local stores are constant throughout the family;
3. 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.

11 Summary

We have examined an approach for verifying concurrent programs.

On one side, we have examined the FQ-algorithm for checking safety of multithreaded programs. We have characterized it in a well-known framework of abstract interpretation. Using this characterization, we have shown the boundary of this algorithm.

On the other side, we have started developing the theory of Cartesian abstraction for multithreaded programs. We have shown two equivalent approaches for abstract fixpoint checking on the abstract and the concrete domain. We have seen that local Cartesian abstraction is polynomial in the number of threads.

Both contributions seem to be first steps in a systematic study of similar abstractions of the state explosion problem.

Acknowledgements

We would like to thank Springer-Verlag for editorial assistance.

References

1. Birkhoff, G., *Lattice Theory*, 3rd ed., Providence, Rhode Island: Amer. Math. Soc., 1967.
2. Blanchet, B., Introduction to Abstract Interpretation, 2002, lecture script, <http://www.di.ens.fr/~blanchet/absint.pdf>
3. Cousot, P., Cousot, R., *Constructive versions of Tarski's fixed point theorems*, Pacific Journal of Mathematics, Vol. 82, No. 1, 1979.
4. Cousot, P., Cousot, R., *Systematic design of program analysis frameworks*, 6th annual ACM symposium on principles of program languages, 1979.
5. Cousot, P., *Partial Completeness of the Abstract Fixpoint checking*, SARA 2000, LNAI 1864, pp. 1-25, 2000.
6. Flanagan, C., Qadeer, S., *Thread-Modular Model Checking*, in T.Ball and S.K. Rajamani (Eds.): SPIN 2003, LNCS 2648, pp. 213-224, 2003, Springer-Verlag Berlin Heidelberg 2003
7. Henzinger, T. A., Jhala, R., Majumdar, R., Qadeer, S., *Thread-modular Abstraction Refinement*, Proceedings of the 15th International Conference on Computer-Aided Verification (CAV), LNCS 2725, Springer-Verlag, pages 262-274, 2003.
8. Holzmann, G. J., *The model checker SPIN*, IEEE Transactions on Software Engineering, 23(5):279-295, May 1997.
9. Kozen, D., *Lower Bounds for Natural Proof Systems*. FOCS 1977, pp.261-262.
10. Muchnik, S. S., Jones, N. D., *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632