Syntactic Regions for Concurrent Programs

Samuel Mimram
École polytechnique

Aly-Bora Ulusoy
École polytechnique

In order to gain a better understanding of the state space of programs, with the aim of making their verification more tractable, models based on directed topological spaces have been introduced, allowing to take in account equivalence between execution traces, as well as translate features of the execution (such as the presence of deadlocks) into geometrical situations. In this context, many algorithms were introduced, based on a description of the geometrical models as regions consisting of unions of rectangles. We explain here that these constructions can actually be performed directly on the syntax of programs, thus resulting in representations which are more natural and easier to implement. In order to do so, we start from the observation that positions in a program can be described as partial explorations of the program. The operational semantics induces a partial order on positions, and regions can be defined as formal unions of intervals in the resulting poset. We then study the structure of such regions and show that, under reasonable conditions, they form a boolean algebra and admit a representation in normal form (which corresponds to covering a space by maximal intervals), thus supporting the constructions needed for the purpose of studying programs. All the operations involved here are given explicit algorithmic descriptions.

1 Introduction

The verification of concurrent programs is notoriously complicated because of the combinatorics involved when performing a naive transposition of the techniques available for sequential programs. Namely, for a program consisting of multiple processes running in parallel, we have to make sure that no problem can arise for whichever respective scheduling of the processes, and the number of resulting execution traces to check is in general exponential in the size of the original program: this is sometimes called the "state-space explosion problem". Moreover, some errors such as the presence of deadlocks are specific to concurrent programs and are not addressed by traditional techniques. In order to tackle these issues, starting in the 90s, a series of theoretical and practical tools have been introduced based on the *geometric semantics* of concurrent programs [11, 12, 7, 6, 4], which assigns to each such program a topological space, such that the possible states of the program can be interpreted as points in this space and an execution as a path which is *directed*, i.e. intuitively respects the direction of time. An important observation is that, in those semantics, two paths which are dihomotopic (can be continuously deformed one to the other while respecting the direction of time) correspond to executions which are equivalent from an operational point of view, and we can thus hope to reduce the state space of the program by considering paths up to dihomotopy. This approach is obviously inspired by the one taken in the algebraic study of spaces (which considers algebraic constructions which are invariant up to homotopy, such as the fundamental and higher homotopy groups), although the situation for programs is made more difficult because one has to take the direction of time in account.

While we believe that this line of research is conceptually enlightening and is very fruitful, the fact that one has to resort to topological spaces in order to study discrete structures such as the ones offered by programs is quite puzzling and it is natural to wonder if the topology is really necessary here. It turns out that it is not, and one of the main goals of this paper is to translate into purely combinatorial terms an important algorithmic construction in geometric semantics: the one of *cubical regions*. We refer the reader to [6, Chapter 5] for a detailed presentation but, roughly, it consists in describing the geometric semantics by covering it with cubes: in such a cube, any two paths are homotopic, and it is thus enough to check only one path in order to perform verification. The starting point of this paper consists in considering that an execution of a program consists in a partial exploration of a "prefix" of this program, that we call here a *position* (on the logical side, similar ideas have been advocated by Girard when defining ludics [8]). A "portion" of the program can then be identified by an interval of positions, which plays the role of the cubes above: it denotes all the positions where we have explored more than the first and less than the second. We can finally reach a convenient description of regions in the program by taking finite unions of such intervals, that we call here *syntactic regions*. We formally define those here and study their properties. Given a set of positions, there are in general multiple regions which describe this set of positions: we show that, under mild assumptions, there is always a canonical region describing a set of positions, the *normal region*, which is in some sense the most economical way of describing it. Moreover, we show that such regions form a boolean algebra (Theorem 31) and provide explicit ways of computing corresponding canonical operations (union, intersection, complement) on syntactic regions, which is useful in practice. Typically, we can compute a representation of the state space of a concurrent program by first computing the region which is forbidden because of the use of mutual exclusion primitives, and then compute the state space as the complement of this region, on which we can use the traditional techniques mentioned above.

In addition to providing us with a better understanding of the "geometry of programs", the aim of the techniques developed is to reduce the size of the state space in order to perform program verification, in a similar vein as partial order reduction (por) [9]. A detailed comparison between the two has been performed both from a practical point of view [5] and a theoretical one [10]: while the two techniques perform similarly on basic examples, geometric techniques sometimes outperform por. Our approach aims at being fully automated and is thus less powerful than techniques based on logic, where some user input or annotations are required (such as the Owicki-Gries method [16], rely/guarantee rules [1], concurrent separation logic [15], etc.). The effectiveness of our approach, which is simpler and easier to implement than previous ones, is illustrated with the online prototype [14], where concrete examples of programs on which it applies are provided (typically, producer-consumer algorithms).

We begin by introducing the programming language considered here as well as an original notion of position on its programs in Section 2, we then define the state space of concurrent programs in Section 3 and explain how it can be described using geometrical regions in Section 4. We generalize the notion of region to arbitrary posets in Section 5, characterize when regions in normal forms have a structure of boolean algebra in Section 6 and finally provide explicit constructions for regions coming from programs in Section 7.

2 Concurrent programs and their positions

In order to abstract away from practical details, we introduce here a simple concurrent imperative programming language. We will only be interested in the control-flow structure and thus the operations we use are not relevant for our matters: we simply suppose fixed a set $\mathscr{A} = \{A, B, \ldots\}$ of *actions*, which can be thought of as the effectful operations of our language, such as modifying a variable or printing a result. The present work would extend to more realistic languages without any difficulty directly related to the main points raised here.

Definition 1. The collection of *programs* P is generated by the following grammar:

$$P,Q \quad ::= \quad A \quad | \quad P\,;Q \quad | \quad P^* \quad | \quad P{+}Q \quad | \quad P\,\|\,Q$$

A program is thus either an action A, or a sequential composition $P\,;Q$ of two programs P and Q, or a conditional branching $P{+}Q$ which will execute either P or Q, or a conditional loop P^* which will execute P a given number of times, or the execution $P\,\|\,Q$ of two subprograms P and Q in parallel. As explained above, we do not take variables into account, so that branching and looping is non-deterministic, but we could handle proper conditional branching and while loops.

A position in a program describes where we are during an execution of it and thus encodes the "prefix" of the program which has already been executed. Formally, we begin by the following definition:

Definition 2. The *pre-positions* p are generated by the following grammar, with $n \in \mathbb{N}$:

$$p,q \quad ::= \quad \bot \quad | \quad \top \quad | \quad p\,;q \quad | \quad p^n \quad | \quad p{+}q \quad | \quad p\,\|\,q$$

Those can be read as: we have not started (resp. we have finished) the execution ($\bot$, resp. $\top$), we are executing a sequence ($p\,;q$), we are in the n-th iteration of a loop (p^n), we are executing a branch of a conditional branching ($p{+}q$) and we are executing two programs in parallel ($p\,\|\,q$). Note that the syntax of positions is essentially the same as the one of programs, except that actions have been replaced by $\bot$ and $\top$ (and loops are "unfolded" in the sense that we keep track of the loop number).

Next, we single out the positions which are valid for a program. For instance, we want that, in a program of the form $P\,;Q$, we can begin executing Q only after P has been fully executed: this means that a position of the form $p\,;q$ with $q \neq \bot$ is valid only when p is $\top$. Similarly, in a conditional branching $P{+}Q$, we cannot execute both subprograms: a position $p{+}q$ is valid only when either p or q is $\bot$.

Definition 3. We write $P \vDash p$ to indicate that a pre-position p is a (valid) *position* of a program P, this predicate being defined inductively by the following rules:

$$\frac{}{P \vDash \bot} \qquad \frac{P \vDash p}{P\,;Q \vDash p\,;\bot} \qquad \frac{P \vDash p}{P{+}Q \vDash p{+}\bot} \qquad \frac{P \vDash p}{P^* \vDash p^n}$$

$$\frac{}{P \vDash \top} \qquad \frac{Q \vDash q}{P\,;Q \vDash \top\,;q} \qquad \frac{Q \vDash q}{P{+}Q \vDash \bot{+}q} \qquad \frac{P \vDash p \qquad Q \vDash q}{P\,\|\,Q \vDash p\,\|\,q}$$

We write $\mathscr{P}(P)$ for the set of positions of a program P.

We can assimilate a position in a program to a possible state during its execution. With this point of view in mind, we can formalize the operational semantics of our programming language as a relation $P \vDash p \to p'$, which can be read as the fact that, in the position p, the program P can reduce in one step and reach the position p'.

Definition 4. The *reduction* relation is defined inductively by

$$\frac{}{A \vDash \bot \to \top}$$

$$\frac{}{P;Q \vDash \bot \to \bot;\bot} \qquad \frac{}{P+Q \vDash \bot \to \bot+\bot} \qquad \frac{}{P^* \vDash \bot \to \bot^0} \qquad \frac{}{P \parallel Q \vDash \bot \to \bot \parallel \bot}$$

$$\frac{P \vDash p \to p'}{P;Q \vDash p;\bot \to p';\bot} \qquad \frac{P \vDash p \to p'}{P+Q \vDash p+\bot \to p'+\bot} \qquad \frac{P \vDash p \to p'}{P^* \vDash p^n \to p'^n} \qquad \frac{P \vDash p \to p' \qquad Q \vDash q}{P \parallel Q \vDash p \parallel q \to p' \parallel q}$$

$$\frac{Q \vDash q \to q'}{P;Q \vDash \top;q \to \top;q'} \qquad \frac{Q \vDash q \to q'}{P+Q \vDash \bot+q \to \bot+q'} \qquad \frac{}{P^* \vDash \top^n \to \bot^{n+1}} \qquad \frac{P \vDash p \qquad Q \vDash q \to q'}{P \parallel Q \vDash p \parallel q \to p \parallel q'}$$

$$\frac{}{P;Q \vDash \top;\top \to \top} \qquad \frac{}{P+Q \vDash \top+\bot \to \top} \qquad \frac{}{P^* \vDash \top^n \to \top} \qquad \frac{}{P \parallel Q \vDash \top \parallel \top \to \top}$$

$$\frac{}{P+Q \vDash \bot+\top \to \top} \qquad \frac{}{P^* \vDash \bot \to \top}$$

The above operational semantics is "very fine-grained" in the sense that it features transitions which are not usually observable, such as $P;Q \vDash \bot \to \bot;\bot$ which corresponds to passing from a state where we have not yet started executing the program to a state where we have started executing a sequence, but not yet its components. The usual "real" actions correspond to executions of the upper left rule $A \vDash \bot \to \top$ which can be interpreted as executing an action A.

Lemma 5. *If $P \vDash p \to p'$ holds then both $P \vDash p$ and $P \vDash p'$ hold.*

Suppose fixed a program P. The *state space* $\mathscr{G}_P$ of this program is the graph whose vertices are the positions of P (Definition 3) and edges are the reductions (Definition 4). An *execution path* of P is a morphism of the free category $\mathscr{G}_P^*$ over the graph $\mathscr{G}_P$. We write $P \vDash \pi : p \to^* p'$ (or simply $\pi : p \to^* q$) to indicate that π is an execution of P from p to q, we write $\pi \cdot \pi'$ for the composition (also called *concatenation*) of two paths $\pi : p \to^* p'$ and $\pi' : p' \to^* p''$, and we write $\varepsilon : p \to^* p$ for the identity execution (also called *empty path*). An execution is *elementary* when it consists of one reduction step. A *global execution* is an execution from $\bot$ and a *total execution* is a global execution to $\top$. We say that an execution π' is a *prefix* of an execution π when there exists an execution π'' such that $\pi = \pi' \cdot \pi''$. A position is *reachable* when there exists a global path $\pi : \bot \to^* p$ with this position as target. The following lemma, together with Lemma 5, formalizes the fact the notion of validity of Definition 3 captures exactly the expected positions.

Lemma 6. *Every position p of P is reachable.*

As customary, we also write $P \vDash p \to^* p'$ (or sometimes simply $p \to^* p'$) when there exists an execution $P \vDash \pi : p \to^* p'$: the resulting relation $\to^*$ on positions of P is the reflexive and transitive closure of the reduction relation $\to$. This relation is induced by a partial order relation, as we now explain.

Definition 7. We write $\leq$ for the smallest reflexive relation on the positions of P such that

$$\frac{}{\bot \leq p} \qquad \frac{p \leq p' \qquad q \leq q'}{p;q \leq p';q'} \qquad \frac{p \leq p' \qquad q \leq q'}{p \parallel q \leq p' \parallel q'} \qquad \frac{p \leq p'}{p^n \leq p'^n}$$

$$\frac{}{p \leq \top} \qquad \frac{p \leq p' \qquad q \leq q'}{p+q \leq p'+q'} \qquad \frac{}{p^m \leq p'^n}$$

for $m < n$.

Proposition 8. *Given two positions p and p' of P, we have $p \leq p'$ if and only if $p \to^* p'$.*

Proposition 9. *The relation $\leq$ is a partial order on the set $\mathscr{P}(P)$ of positions of P.*

Proposition 10. *The partial order $\leq$ on $\mathscr{P}(P)$ is a bounded lattice, with $\bot$ and $\top$ as smallest and largest elements, with supremum being determined by*

$$\begin{array}{lll} (p\,;q) \vee (p'\,;q') = (p \vee p')\,;(q \vee q') & p^n \vee p'^n = (p \vee p')^n & (p+\bot) \vee (p'+\bot) = (p \vee p')+\bot \\ (p \parallel q) \vee (p' \parallel q') = (p \vee p') \parallel (q \vee q') & p^m \vee p'^n = p'^n & (\bot+q) \vee (\bot+q') = \bot+(q \vee q') \\ & & (p+\bot) \vee (\bot+q) = \top \end{array}$$

for $m < n$, and where we suppose $(p,q) \neq (\bot,\bot)$ in the lower right rule. The infimum admits a similar description.

We recall that a poset $(X,\leq)$ is a *well-order* when, for every infinite sequence $(x_i)_{i\in\mathbb{N}}$ of elements, there are indices $i < j$ such that $x_i \leq x_j$. This is equivalent to requiring that the poset is both well-founded (every infinite decreasing sequence of elements is eventually stationary) and such that every antichain (set of pairwise incomparable elements) is finite. The following will be useful:

Proposition 11. *The poset $\mathscr{P}(P)$ is a well-order.*

Proof. The proof proceeds by induction on P. The inductive cases are easily deduced from the fact that well-orders are closed under products, coproducts and contain $(\mathbb{N},\leq)$. □

3 Concurrent programs with mutexes and their state space

In practice, in order to avoid unspecified behaviors due to concurrency, programs use primitive operations such as mutexes [3] with the aim of preventing that two subprograms access simultaneously to a shared resource such as memory (typically, the result of two concurrent writings at a same memory location is unspecified in many languages). In order to account for this, we suppose fixed a set $\mathscr{M}$ of *mutexes* such that, for every $a \in \mathscr{M}$, there are two associated actions $\mathtt{P}_a$ and $\mathtt{V}_a$ in $\mathscr{A}$, respectively corresponding to *taking* and *releasing* the mutex a, as defined in [3]. A mutex will be such that it can be can be taken by at most one subprogram: if another subprogram tries to take an already taken resource, it will be "frozen" until the mutex is available again, i.e. the first subprogram releases the mutex. This description is formalized below on the operational semantics, see also [6, Chapter 3].

The consumption of mutexes by an execution is kept track of by considering functions in $\mathbb{Z}^{\mathscr{M}}$ which to every mutex associate the number of times it has been taken or released (where releasing is considered as the opposite of taking). Given two functions $\mu,\mu' \in \mathbb{Z}^{\mathscr{M}}$, we write $\mu+\mu'$ for their pointwise sum, i.e. $(\mu+\mu')(a) = \mu(a)+\mu'(a)$ for $a \in \mathscr{M}$, and similarly for the opposite $-\mu$ of a function μ. Given $a \in \mathscr{M}$, we write $\delta_a \in \mathbb{Z}^{\mathscr{M}}$ for the function such that $\delta_a(a) = 1$ and $\delta_b(a) = 0$ for $b \neq a$, we also write $\underline{0}$ for the constant function equal to 0.

Definition 12. Given an execution $P \vDash \pi : p \to^* p'$, we write $[\![P \vDash \pi : p \to^* p']\!] : \mathscr{M} \to \mathbb{Z}$ (or sometimes simply $[\![\pi]\!]$) for its *consumption* of mutexes. This function is defined for elementary executions by

- $[\![\mathtt{P}_a \vDash \pi : \bot \to \top]\!] = \delta_a$, for $a \in \mathscr{M}$,
- $[\![\mathtt{V}_a \vDash \pi : \bot \to \top]\!] = -\delta_a$, for $a \in \mathscr{M}$,
- for each reduction π, different from the two above, deduced with a rule of Definition 4 with no premise we have $[\![\pi]\!] = \underline{0}$,

- for each reduction π deduced with a rule of Definition 4 with one reduction π' as premise, we have $[\![\pi]\!] = [\![\pi']\!]$,

and extended as an action of the category of executions on the monoid of consumptions, i.e. $[\![\varepsilon]\!] = \underline{0}$ and $[\![\pi \cdot \pi']\!] = [\![\pi']\!] + [\![\pi]\!]$.

Example 13. Writing π for any total execution of the program $\mathtt{P}_a\,;(\mathtt{P}_a \parallel \mathtt{V}_b)$, with $a \neq b$, we have $[\![\pi]\!](a) = 2$, $[\![\pi]\!](b) = -1$ and $[\![\pi]\!](c) = 0$ for $c \neq a$ and $c \neq b$.

A global execution π is *valid* when for every prefix π' of π, and any mutex $a \in \mathscr{M}$, we have $0 \leq [\![\pi']\!](a) \leq 1$. Such an execution is namely compatible with the semantics of mutexes in the sense that

- no mutex is taken twice (without having been released in between),
- no mutex is released without having been taken first.

A program is *conservative* when any two executions $\pi : p \to^* p'$ and $\pi' : p \to^* p'$ with the same source and the same target have the same resource consumption: $[\![\pi]\!] = [\![\pi']\!]$. In the following, unless otherwise stated, we assume that all the programs we consider are conservative: it is often satisfied in practice and simplifies computations because we can consider validity of positions instead of executions. This assumption is classical in geometric semantics [6] and is based on the observation that the use of mutexes is quite costly (because they restrict parallelism) and therefore usually restricted to small and ”local” portions of the code. This is the reason why it is usually satisfied, at least up to simple rewriting of the code (more complex code can also be handled by “duplicating” some of the positions so that the program becomes conservative). This condition can easily be tested as follows.

Definition 14. The *consumption* of a program P is the partial function $\Delta(P) : \mathscr{M} \to \mathbb{Z}$ defined by induction on P by

$$\Delta(\mathtt{P}_a) = \delta_a \qquad \Delta(\mathtt{V}_a) = -\delta_a \qquad \Delta(A) = \underline{0} \qquad \Delta(P;Q) = \Delta(P \parallel Q) = \Delta(P) + \Delta(Q)$$

and

$$\Delta(P{+}Q) = \Delta(P) \quad \text{if } \Delta(P) = \Delta(Q) \qquad\qquad \Delta(P^*) = \underline{0} \quad \text{if } \Delta(P) = \underline{0}$$

Note that, because of the side conditions in the two last cases, $\Delta(P)$ is not always well defined, e.g. for $\mathtt{P}_a^*$ or $\mathtt{P}_a{+}\mathtt{P}_b$.

Proposition 15. *A program P is conservative if and only if $\Delta(P)$ is well-defined and, in this case, we have $\Delta(P) = [\![\pi]\!]$ for any total execution π of P.*

The above proposition gives a simple criterion to check for conservativity, by induction on programs. It is applicable even when the set $\mathscr{M}$ of mutexes is infinite because it is not difficult to show that, for any program P, $\Delta(P)$ is almost everywhere null when defined, i.e. the set $\{a \in \mathscr{M} \mid \Delta(P)(a) \neq 0\}$ is finite, so that the computations on consumption can easily be implemented in practice. For conservative programs, it makes sense to consider the resource consumption at a position (as opposed to along an execution):

Definition 16. Given a conservative program, we define the *consumption* $[\![p]\!] : \mathscr{M} \to \mathbb{Z}$ of a position p as $[\![p]\!] = [\![\pi]\!]$ for some global path $\pi : \bot \to p$.

This definition makes sense because there is at least one such path π by Lemma 6 and it does not depend on the choice of the path by the assumption of being conservative. This enables us to characterize valid executions as follows. We say that an execution $\pi : p \to^* p'$ *visits* a position q when q is the target of some prefix $\pi' : p \to^* q$ of π. We say that a position p is *valid* if it satisfies $0 \leq [\![p]\!](a) \leq 1$ for every mutex $a \in \mathscr{M}$.

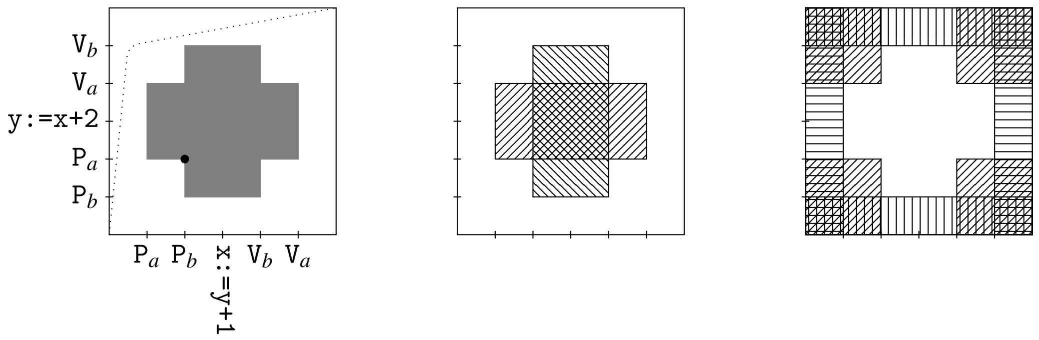


Figure 1: Geometric semantics of a simple program.

Proposition 17. *A global execution π is valid if and only if every position p it visits is valid.*

This motivates defining the *pruned state space* $\overline{\mathscr{G}}_P$ of P as the subgraph obtained from $\mathscr{G}_P$ by removing every edge between invalid vertices, and all vertices whose incident edges have all been removed. Writing $\overline{\mathscr{G}}^*_P$ for the free category it generates, which is a subcategory of $\mathscr{G}^*_P$, we have

Proposition 18. *The morphisms of $\overline{\mathscr{G}}^*_P$ are are precisely the valid executions of P.*

4 Geometric semantics

We now briefly recall (or rather illustrate) the geometric semantics of concurrent programs, and relate it to the previous point of view on programs. We refer to the textbook [6] as well as [13] for further reference on the subject.

In order to take a concrete example, we suppose here only that our actions allow for traditional manipulations of variables, and consider the program

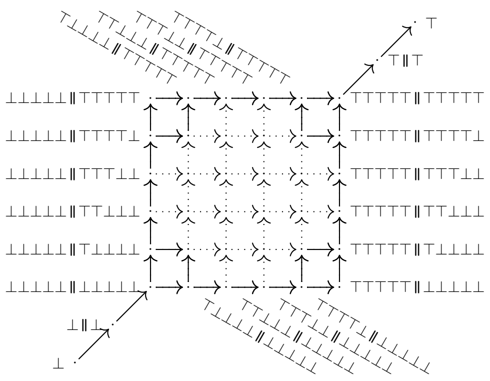
$$(\mathtt{P}_a \;;\; \mathtt{P}_b \;;\; \mathtt{x := y+1} \;;\; \mathtt{V}_b \;;\; \mathtt{V}_a) \parallel (\mathtt{P}_b \;;\; \mathtt{P}_a \;;\; \mathtt{y := x+2} \;;\; \mathtt{V}_a \;;\; \mathtt{V}_b)$$

consisting of two processes executed in parallel. Here, we should think of a and b as protecting the variables x and y respectively: the program applies the discipline that whenever a variable is used, the corresponding mutex is taken beforehand and released afterward, in order to avoid any concurrent access to it. The idea behind geometric semantics is that to such a program we should associate the topological space on the left of Fig. 1, which consists of a square from which the grayed region has been removed (this is called the *forbidden region*). The horizontal axis corresponds to the execution of the first process and the vertical one to the second process (we have indicated the points corresponding to each instruction of the processes in abscissa and ordinate in order to make this clear). A point in this space can thus roughly be assimilated to a position in the program and the grayed removed area corresponds to removing forbidden positions. A (continuous) path in this space is said to be *directed* when it is increasing with respect to each component: such a path corresponds to an execution of the program. We sometimes say a *dipath* for a directed path. For instance, the dotted path on the left of Fig. 1 is directed and corresponds to a total execution where the second process gets wholly executed before the first one does. Finally, two dipaths are *dihomotopic* when the first can be continuously deformed to the second while going through directed paths only. In general, the definition of the geometric semantics is more involved than the above example shows (in particular, the notion of direction is subtle in presence of loops) and is detailed in the aforementioned references.

Let us now mention two main applications of geometric semantics. Firstly, under simple *coherence* assumptions, it can be shown that two dihomotopic dipaths have the same effect on the state: in order to ensure the absence of errors in a program (e.g. we are never dividing by 0), it is thus sufficient to use traditional techniques (e.g. abstract interpretation [2]) for one representative in each equivalence class. Secondly, it can be helpful to detect problems specific to concurrency in programs. For instance, the black dot in the figure on the left is a point which is not the final point (the upper right one) and is the source of no non-trivial directed path: this indicates the presence of a *deadlock* in the program. Namely, if the first process executes $\mathtt{P}_a$ then $\mathtt{P}_b$ and the second process executes $\mathtt{P}_b$ then $\mathtt{P}_a$ the program is locked and cannot proceed any further: the first process is waiting for the second to release the mutex b and the second process is waiting for the first to release the mutex a.

We do not develop these techniques much further here – they have already been elsewhere – and concentrate on the following question: how can we represent this geometric semantics in practice? For programs consisting of n processes in parallel, such as the one above, where each process consists of a sequence of actions, the geometric semantics will be an n-dimensional cube from which a finite number of n-dimensional cubes have been carved out, forming the forbidden region (more general cases are handled in [13]). For instance, in our example, the forbidden region consists of two cubes as shown in the figure in the middle of Fig. 1. This motivates investigating the notion of (*cubical*) *region*, which is a portion of the global cube obtained as a finite union of cubes. Interestingly, the geometric semantics, which is the complement of the forbidden region, can also be described as a union of cubes (8 in our example), and is thus a region, as pictured on the right of Fig. 1. In particular, by convexity, any two dipaths with the same endpoints within a cube are dihomotopic, which helps computing representatives in dihomotopy classes.

The state space associated to the above program is



(for simplicity, we have omitted some intermediate positions such as $(\bot\,;\bot) \parallel \bot$, also we have omitted writing "$;$" in the positions). The dotted edges are those which have to be removed in order to obtain the pruned state space. One can see that, up to some minor details such as the added positions at the beginning and at the end, the pruned graph can be thought of as an "algebraic counterpart" of the geometric semantics, which motivates the investigation of performing the computations directly on this representation, instead of going though an arbitrary encoding into spaces. In particular, we define and study here an abstract axiomatization of regions, which applies to such graphs.

5 Regions of posets

In this section we define our representation of the state-space of programs, replacing (maximal) cubical regions in geometric semantics [6] by (normal) *syntactic regions*. We show that, under mild assumptions, these regions satisfy the same fundamental properties as cubical regions (forming a boolean algebra, supporting the existence of canonical representatives, etc.) and provide explicit ways of computing corresponding canonical operations on syntactic regions. To the best of our knowledge all the constructions performed here are new, and based on the original notion of *position* introduced in Section 2.

5.1 Intervals

Suppose fixed a poset $(X,\leq)$. An *interval* (x,y) in this poset is a pair of elements such that $x \leq y$. Given an interval $I=(x,y)$, we write $[I]=[x,y]=\{z \in X \mid x \leq z \leq y\}$ for the set of points it contains, also called its *support*. We write $z \in I$ instead of $z \in [I]$: we have $z \in (x,y)$ iff $x \leq z \leq y$. We also write $I \subseteq J$ instead of $[I] \subseteq [J]$: we have $(x,y) \subseteq J$ iff $x \in J$ and $y \in J$. Finally, we denote by $\mathscr{I}(X)$ the poset of intervals of X.

5.2 Regions

A *region* R is a set of intervals: we write $\mathscr{R}(X)=\mathfrak{P}(X \times X)$ for the set of regions of X, where $\mathfrak{P}(-)$ denotes the powerset of a given set. A region is *finite* when it consists of a finite number of intervals. The *support* of a region R is the set $[R]=\bigcup_{I\in R}[I]$ of points it contains.

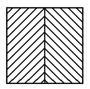
Two regions are *equivalent* when they have the same support. Such regions are different ways of describing the same set of points: in order to be able to correctly manipulate regions, we should be able to decide when two regions are equivalent, and in order to be efficient, we should find the most compact representation of a region in its equivalence class. Intuitively, the “best” region with a given support $Y \subseteq X$ consists of all the maximal intervals (wrt $\subseteq$) contained in Y. We will call this region the *normal form* for a region R and say that a region is in normal form when it is equal to its own normal form. In order to formalize this, we introduce the following preorder on regions: we write $\preceq$ for the pre-order on regions in $\mathscr{R}(X)$ defined by

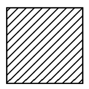
$$R \preceq S \qquad \text{iff} \qquad \forall I \in R.\, \exists J \in S.\, I \subseteq J$$

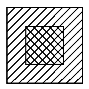
When $R \preceq S$ and R and S are equivalent, we think of S as being a “more economic way” of describing the same support, because it uses bigger intervals: every interval of R is contained in one of S.

Lemma 19. *The functions* $[-]:\mathscr{R}(X)\to\mathfrak{P}(X)$ *and* $\mathscr{I}:\mathfrak{P}(X)\to\mathscr{R}(X)$ *are increasing and* $[-]$ *is left adjoint to* $\mathscr{I}$.

Example 20. Consider the set $X=[0,1]^2$ equipped with the usual partial order, and consider the three following regions on it, whose support is X:


$R_1=\{[0,\frac{1}{2}]\times[0,1],[\frac{1}{2},1]\times[0,1]\}$


$R_2=\{[0,1]\times[0,1]\}$


$R_3=R_2\cup\{[\frac{1}{4},\frac{3}{4}]\times[\frac{1}{4},\frac{3}{4}]\}$

We have $R_1 \prec R_2$ and $R_3 \preceq R_2$, as well as $R_2 \preceq R_3$.

In the above example, the region R_2 is clearly the most parsimonious way to represent the whole space, in the sense explained above, and is a maximal element with respect to the order. However, there are many other maximal elements such as R_3 (or, in fact, any other region obtained by adding intervals to R_2: the relation $\preceq$ is not antisymmetric). This motivates the introduction of the following refined order on regions, which is such that $R_1 < R_3 < R_2$:

Definition 21. We define the relation $\leq$ on $\mathscr{R}(X)$ by $R \leq S$ if and only if $R \preceq S$, and $S \preceq R$ implies $S \subseteq R$ (i.e. every interval of S belongs to R).

Lemma 22. *The relation $\leq$ is a partial order.*

Lemma 23. *The support function $[-] : \mathscr{R}(X) \to \mathfrak{P}(X)$ is increasing if we equip the first with $\leq$ and second with $\subseteq$ as partial orders.*

We expect that the "best description" of a subset of X by a region is given by a right adjoint $N : \mathfrak{P}(X) \to \mathscr{R}(X)$ to the support function, which to a subset Y of X associates the normal region describing it. Namely, the adjoint functor theorem indicates that if the right adjoint exists it should satisfy

$$N(Y) = \bigvee\{R \in \mathscr{R}(X) \mid [R] \subseteq Y\} \tag{1}$$

However, such an adjoint is not always well-defined, as illustrated in Example 25 below. The *normal form* of a region R is, when defined, the region $N([R])$, where N is defined by the formula (1) above. We say that a region is *normalizable* when it admits a normal form, and *in normal form* when it is further equal to its normal form.

Lemma 24. *Given $Y \subseteq X$ such that $N(Y)$ is defined, one has $[N(Y)] = Y$.*

Example 25. Consider the set $X = [0,1] \subseteq \mathbb{R}$ equipped with the usual order. We claim that the subset $Y = X \setminus \{1\}$ does not have a normal form. By contradiction, suppose that the region $N(Y)$ is well-defined. By the above Lemma 24, $[N(Y)] = Y = [0,1[$. Given $I \in N(Y)$, there exists ε, η with $0 \leq \varepsilon \leq 1-\eta < 1$ such that $I = [\varepsilon, 1-\eta]$ and one easily checks that the region R obtained from $N(Y)$ by removing this interval and replacing it by $[0, 1-\eta/2]$ is such that $[R] = [N(Y)] = Y$ and $N(Y) < R$, contradicting (1). A very similar situation can be observed in the case of programs, by considering the program $P = A^*$ for some action A. We write $X = \mathscr{P}(P)$ for its poset of positions:

$$\begin{tikzcd}
\overset{\perp}{\cdot} \arrow[r] & \overset{\perp^0}{\cdot} \arrow[r] & \overset{\top^0}{\cdot} \arrow[r] & \overset{\perp^1}{\cdot} \arrow[r] & \overset{\top^1}{\cdot} \arrow[r] & \overset{\perp^2}{\cdot} \arrow[r] & \cdots\cdots\cdots \overset{\top}{\cdot}
\end{tikzcd}$$

The subset $Y = X \setminus \{\top\}$ does not have a normal form for similar reasons as above.

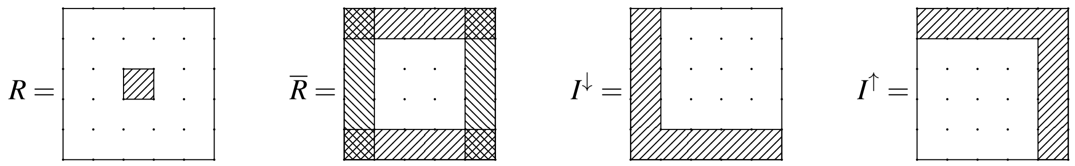
Remark 26. In order to accommodate with situations such as in previous example, one could think of allowing (semi-)open intervals in addition to closed ones in regions. However, the operations on those quickly become very difficult to handle because it turns out that, in the case of programs of the form $P \parallel Q$, we need to be able to specify whether bounds of intervals are open or not for each component of the parallel composition.

6 The boolean algebra of finitely complemented regions

Given a set $Y \subseteq X$, we write $\bar{Y} = X \setminus Y$ for its *complement*. For practical applications, given a region R in $\mathscr{R}(X)$, we need to be able to compute a region $\overline{R}$ which covers the complement of the region, i.e. a region $\overline{R}$ such that $[\overline{R}] = \overline{[R]}$. Typically, we have explained in Section 3 that the pruned state space is obtained as the complement in the state space of the forbidden region. Moreover, even when the region R is not in normal form, we are able to compute the region $\overline{R}$ in normal form, which suggests that we should be able to compute the normal form of a region R as $N([R]) = \overline{\overline{R}}$. Performing these computations will also involve computing intersections and unions of regions, so that we will show that – under suitable hypothesis – regions which admit a normal form have the structure of a boolean algebra, providing explicit algorithmic constructions for the corresponding operations. This generalizes the situation considered in [6] (which is limited to regions which are subsets of $\mathbb{R}^n$) and in [13] (which is limited to products of graphs). In order to ensure that our constructions are applicable in practice, we only consider regions which are finite in the following (and showing that finiteness is preserved by our constructions will be non-trivial). In particular, by a "normal form", we always mean a finite region in normal form.

6.1 The complement of an interval

We begin by investigating the computation of the region corresponding to the complement of an interval. As an illustration, consider the space $X = [0,5]^2 \subseteq \mathbb{N}^2$ and $R = \{I\}$ with $I = [(2,2),(3,3)]$, as pictured on the left



The normal form of its complement is the region

$$\overline{R} = \{[\bot, y_1], [\bot, y_2], [x_1, \top], [x_2, \top]\}$$

with $\bot = (0,0)$, $\top = (5,5)$, $y_1 = (1,5)$, $y_2 = (5,1)$, $x_1 = (0,4)$, $x_2 = (4,0)$. Here, it should be noted that $\overline{R} = I^{\downarrow} \cup I^{\uparrow}$ where $I^{\downarrow}$ (resp. $I^{\uparrow}$) is the set of elements of X which are not above $(2,2)$ (resp. below $(3,3)$), nor in fact above any element of I. Moreover, the points y_1 and y_2 are the maximal elements of the set $I^{\downarrow}$ and, similarly, the points x_1 and x_2 are the minimal element of $I^{\uparrow}$. We can generalize the situation as follows.

Given a set $Y \subseteq X$, its *lower closure* $\downarrow Y$ and *lower complement* $Y^{\downarrow}$ are respectively the sets

$$\downarrow Y = \{x \in X \mid \exists y \in Y.\, x \leq y\} \qquad\qquad Y^{\downarrow} = \{x \in X \mid \forall y \in Y. x \not\geq y\}$$

The *upper closure* $\uparrow Y$ and *lower complement* $Y^{\downarrow}$ of Y are defined dually. We write $\max Y$ for the set of maximal elements of Y, and $\min Y$ for the set of minimal elements. The set Y is *finitely lower generated* when there exists a finite set Y' such that $Y = \downarrow Y'$ (the notion of *finitely upper generated* is defined dually). By extension, we say that an element x of X is finitely lower (resp. upper) generated when $\{x\}$ is.

Lemma 27. *If Y is finitely lower generated then* $\max Y$ *is finite and we have $Y = \downarrow \max Y$.*

We say that a set $Y \subseteq X$ is *finitely lower* (resp. *upper*) *complemented* when $Y^{\downarrow}$ (resp. $Y^{\uparrow}$) is finitely lower (resp. upper) generated. By extension, we say that an element x of X is finitely lower (resp. upper) complemented when $\{x\}$ is. We say that Y is *finitely complemented* when it is both finitely lower and upper complemented. We can finally characterize intervals which admit a complement in normal form as follows:

Proposition 28. *Given a bounded lattice X and I an interval, the region $R = \{I\}$ has a complement in normal form if and only if $[I]$ is finitely complemented. In this case, the normal form of its complement is*

$$\overline{R} = \{[\bot, y] \mid y \in \max(I^{\downarrow})\} \cup \{[x, \top] \mid x \in \min(I^{\uparrow})\}$$

6.2 The complement of a region

Our aim is now to generalize Proposition 28, and characterize normalizable regions (as opposed to intervals) which admit a complement in normal form. The condition which emerged in order to capture such situations is the following one:

Definition 29. A poset $(X, \leq)$ is *finitely complemented* if

1. given a finitely lower complemented element $x \in X$, every element of $\max(\{x\}^{\downarrow})$ is finitely upper complemented,
2. given a finitely upper complemented element $x \in X$, every element of $\min(\{x\}^{\uparrow})$ is finitely lower complemented.

Remark 30. In the case where X is a well-order, every upwards closed subset is necessarily finitely upper generated: the set $\min X$ of minimal elements of X generates X because it is well-founded, and is finite because it is an antichain. The first condition is thus always satisfied.

We suppose fixed an ambient bounded lattice X, in which the construction will be performed. For technical reasons, it will be convenient to suppose that X is also well-ordered, which, by Proposition 11, is a reasonable restriction for the applications we have in mind. We write

$$\mathscr{N}(X) = \{Y \subseteq X \mid \text{both } N(Y) \text{ and } N(\overline{Y}) \text{ exists and are finite}\}$$

for the poset (under inclusion) of *normal supports* (i.e. supports of regions in normal forms, whose complements are also supports of regions in normal forms). Our goal is to show the following theorem:

Theorem 31. *The poset $\mathscr{N}(X)$ is a boolean algebra if and only if X is finitely complemented.*

One of the implications is easily shown:

Proposition 32. *The left-to-right implication of Theorem 31 holds.*

Proof. By contraposition, suppose that X is not finitely complemented. By Remark 30, the condition (i) in the Definition 29 is always satisfied, and therefore (ii) has to be falsified. This means there exists an upper complemented element $x \in X$ such that there exists an element $y \in \min(\{x\}^{\uparrow})$ such that y is not finitely lower generated. We show below that both $[\bot, y]$ and $\overline{[\bot, x]}$ belong to $\mathscr{N}(X)$. But their intersection is not in $\mathscr{N}(X)$: the set $\mathscr{N}(X)$ is not closed under intersections and thus not a boolean algebra. Namely, an easy computation shows $[\bot, y] \cap \overline{[\bot, x]} = [y, y]$ and we have that $[y, y] \notin \mathscr{N}(X)$ by Proposition 28, because y is not finitely lower complemented.

It remains to be shown that $[\bot, y] \in \mathscr{N}(X)$. Since y is upper complemented and $\bot$ is trivially lower complemented, we have that the interval $I = (\bot, y)$ is finitely complemented (by definition). Writing $R = \{I\}$, the region R is trivially in normal form, and has a complement in normal form by Proposition 28, and therefore I belongs to $\mathscr{N}(X)$. The proof that $\overline{[\bot, x]} \in \mathscr{N}(X)$ is similar. □

We say that a region R is *finitely complemented* if it contains only intervals whose support is finitely complemented in the sense of Section 6.1. Beware that this definition does not state that the support of the region should be finitely complemented. We write

$$\mathscr{F}(X) = \{Y \subseteq X \mid Y = [R] \text{ for some finitely complemented region } R\}$$

We also write

$$\mathscr{R}_{\mathscr{F}}(X) = \{R \in \mathscr{R}(X) \mid R \text{ is finitely complemented}\}$$

so that the elements of $\mathscr{F}(X)$ are the supports of regions in $\mathscr{R}_{\mathscr{F}}(X)$. The plan of our proof for the missing implication of Theorem 31 is as follows: we first show that $\mathscr{F}(X)$ forms a boolean algebra by explicitly constructing the required operations on finitely complemented regions (Proposition 36) and then show that $\mathscr{F}(X)$ is isomorphic to $\mathscr{N}(X)$ (Proposition 37).

In the rest of this section, we suppose that the poset X is finitely complemented. Given two intervals $I = (x,y)$ and $I' = (x',y')$, we write $(x,y) \cap (x',y') = (x \vee x', y \wedge y')$ for their intersection, which is always their infimum (wrt $\subseteq$ order) when defined (i.e. $x \vee x' \leq y \wedge y'$).

Definition 33. We define the following operations on regions R,S in $\mathscr{R}_{\mathscr{F}}(X)$:

- union: $R \cup_N S = R \cup S$
- intersection: $R \cap_N S = \{I \cap J \mid I \in R,\, J \in S \text{ and } I \cap J \text{ is defined}\}$
- complement: $\overline{R}^N = \bigcap_{N\ (x,y)\in R} \left(\{(\bot,y') \mid y' \in \max(\{x\}^{\downarrow})\} \cup \{(x',\top) \mid x' \in \min(\{y\}^{\uparrow})\}\right)$

Lemma 34. *The above operations are well-defined on $\mathscr{R}_{\mathscr{F}}(X)$.*

Proof. The most subtle is intersection. It is shown by proving that finitely lower (resp. upper) complements of elements of X are stable by $\vee$ (resp. $\wedge$). □

Lemma 35. *The above operations are compatible with the corresponding ones on supports: for regions $R,S \in \mathscr{R}_{\mathscr{F}}(X)$, we have*

$$[R \cap_N S] = [R] \cap [S] \qquad [R \cup_N S] = [R] \cup [S] \qquad [\overline{R}^N] = \overline{[R]}$$

The finitely complemented regions $\mathscr{R}_{\mathscr{F}}(X)$ thus form a sub-boolean algebra of $\mathfrak{P}(X)$:

Corollary 36. *The set $\mathscr{F}(X)$ is a boolean algebra.*

Proposition 37. *Finitely complemented supports coincide with normal ones, i.e. we have $\mathscr{F}(X) = \mathscr{N}(X)$.*

Proof. $\mathscr{F}(X) \subseteq \mathscr{N}(X)$. Given $Y \in \mathscr{F}(X)$, there exists a region $R \in \mathscr{R}_{\mathscr{F}}(X)$ such that $[R] = Y$. By Lemma 34, its complement $\overline{R}^N$ belongs to $\mathscr{R}_{\mathscr{F}}(X)$, since it can be computed using a finite number of unions and intersections of finitely complemented regions. Then, it is easy to prove $\max \overline{R}^N = N(\overline{[R]}) = N(\overline{Y})$. Applying this twice gives the normal form of R, and thus of Y.

$\mathscr{F}(X) \supseteq \mathscr{N}(X)$. Suppose given $Y \in \mathscr{N}(X)$. We are going to show $\overline{Y} \in \mathscr{F}(X)$ and by Proposition 37, we will conclude $Y \in \mathscr{F}(X)$. We define an extension $\overline{-}^{\infty} : \mathscr{R}(X) \to \mathscr{R}(X)$ of $\overline{-}^N$ for regions that are not finitely complemented by

$$\overline{R}^{\infty} = \bigcap_{N\ (s,t)\in R} \{(\bot,x) \mid x \in s^{\downarrow}\} \cup_N \{(x,\top) \mid x \in \max t^{\uparrow}\}$$

and we show that $N(\overline{Y}) \subseteq \overline{N(Y)}^{\infty}$ and $\overline{N(Y)}^{\infty} \in \mathscr{R}_{\mathscr{F}}(X)$, by Remark 30. Thus $N(\overline{Y}) \in \mathscr{R}_{\mathscr{F}}(X)$, i.e. $\overline{Y} \in \mathscr{F}(X)$ and finally $Y \in \mathscr{F}(X)$. □

We have thus shown the other implication of Theorem 31:

Corollary 38. *If X is finitely complemented, the set $\mathscr{N}(X)$ is a boolean algebra.*

This thus shows that, in a finitely complemented poset, we can implement the usual boolean operations on regions while preserving the property of being normalizable.

7 Syntactic regions

In the case of syntactic regions, i.e. regions on the poset of positions of a program, the operations of boolean algebra can be effectively implemented, by induction on the structure of programs, following Definition 33. Namely,

- the union of regions is immediate to implement,
- the supremum and infimum of positions can be computed following Proposition 10, from which we can compute the intersection of regions,
- we can compute the generators of the complement, from which we can compute the complement of regions.

Let us detail the last point. Given a position p of a program P, we define, by induction on P, the following set $\inf_P(p)$ of positions of P:

$$\begin{array}{lll} \inf_P(\bot) = \emptyset & \inf_{P;Q}(\top) = \{\top;\top\} & \inf_{P+Q}(\top) = \{\top+\bot, \bot+\top\} \\ \inf_A(\top) = \{\bot\} & \inf_{P\parallel Q}(\top) = \{\top \parallel \top\} & \inf_{P^*}(\top) = \emptyset \end{array}$$

$$\inf_{P;Q}(p;q) = \begin{cases} \{\bot\} & \text{if } p = q = \bot \\ \{p';\bot \mid p' \in \inf_P(p)\} & \text{if } p \neq \bot \text{ and } q = \bot \\ \{\top;q' \mid q' \in \inf_Q(q)\} & \text{if } p = \top \text{ and } q \neq \bot \end{cases}$$

$$\inf_{P+Q}(p+q) = \begin{cases} \{\bot\} & \text{if } p = q = \bot \\ \{p'+q \mid p' \in \inf_P(p)\} \cup \{p+\top\} & \text{if } p \neq \bot \\ \{p+q' \mid q' \in \inf_Q(q)\} \cup \{\top+q\} & \text{of } q \neq \bot \end{cases}$$

$$\inf_{P\parallel Q}(p \parallel q) = \begin{cases} \{\bot\} & \text{if } p = q = \bot \\ \{p' \parallel \top \mid p' \in \inf_P(p)\} \cup \{\top \parallel q' \mid q' \in \inf_Q(q)\} & \text{if } p \neq \bot \text{ or } q \neq \bot \end{cases}$$

$$\inf_{P^*}(p^n) = \begin{cases} \{\bot\} & \text{if } n = 0 \text{ and } p = \bot \\ \{\top^{n-1}\} & \text{if } n > 0 \text{ and } p = \bot \\ \{p'^n \mid p' \in \inf_P(p)\} & \text{if } p \neq \bot \end{cases}$$

Proposition 39. *Given a position p of a program P, the set $\inf_P(p)$ is a well-defined set of positions and we have $\inf_P(p) = \max(p^\downarrow)$.*

Similarly, given a position p of a program P, one can define by induction on P a set $\sup_P(p)$ of positions of P such that $\sup_P(p) = \min(p^\uparrow)$. We can finally show that the poset of positions of a programs satisfy the conditions of previous section:

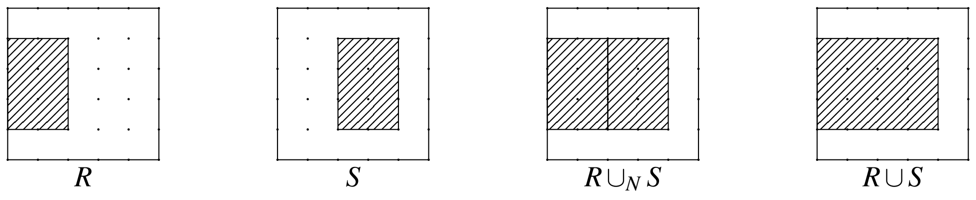
Proposition 40. *Given a program P, its poset of positions $\mathscr{P}(P)$ is a finitely complemented well-ordered lattice.*

The operations do not in general preserve the property of being normal for regions, but Theorem 31 and Proposition 40 however ensure that the property of being normalizable is preserved, and the normal form of a region can be computed as follows:

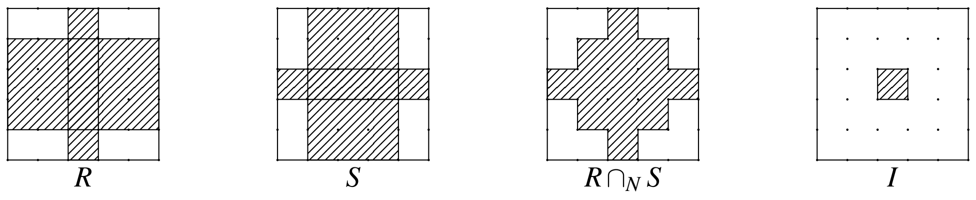
Proposition 41. *With the implementation of operations described above, the normal form of a region R is $N(R) = \max(\overline{\overline{R}})$, i.e. it can be obtained by computing twice the complement of R and only keeping intervals which are maximal wrt inclusion.*

Proof. It can be observed that the definition of the complement is such that it contains all the maximal intervals. □

Example 42. Let us illustrate the fact that the operations defined above do not preserve normality (again, they only preserve normalizability), consider the following examples. With the region R and S below, the region $R \cup_N S$ is not normal (the normal form is show on the right):



Similarly, with the region R and S below, the region $R \cap_N S$ is not normal because it contains the interval I pictured on the right



(the normal form contains 3 intervals which do not include I).

8 Future work

A toy implementation of the computations described in this paper can be tested online at [14]. It allows computing the forbidden region, the state space (called there the *fundamental region*) and the deadlocks of a program with loops (we also plan to implement of further analysis of programs, handling values with abstract domains as explained in the introduction). The positions for loops are defined there *without* unfolding: this allows handling the case of forbidden regions within loops, but the theory is more involved (the execution relation on position does not induce a partial order anymore) and left for future work. Various practical examples of concurrent programs are given on the website and the reader is welcome to try out some more of his own. We plan to investigate, in a near future, the investigation of the combination of this approach with abstract interpretation techniques in order to be able to meaningfully handle domains of values. Finally, we also plan to perform a formalization (in Agda) of the theory developed there in order to make sure that no corner case is omitted (as it can be observed in Section 7, the operations are defined by case analysis, requiring to distinguish many possibilities and the situation is even worse in generalizations).

References

[1] Joey W Coleman & Cliff B Jones (2007): *A structural proof of the soundness of rely/guarantee rules*. *Journal of Logic and Computation* 17(4), pp. 807–841, doi:10.1093/logcom/exm030.

[2] Patrick Cousot & Radhia Cousot (1977): *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, doi:10.1145/512950.512973.

[3] Edsger Wybe Dijkstra (1965): *Solution of a problem in concurrent programming control*. Communications of the ACM 8(9), p. 569, doi:10.1007/978-3-642-48354-7_10.

[4] Lisbeth Fajstrup, Éric Goubault, Emmanuel Haucourt, Samuel Mimram & Martin Raussen (2012): *Trace spaces: An efficient new technique for state-space reduction*. In: European Symposium on Programming, Springer, pp. 274–294, doi:10.1007/978-3-642-28869-2_14.

[5] Lisbeth Fajstrup, Éric Goubault, Emmanuel Haucourt, Samuel Mimram & Martin Raussen (2012): *Trace Spaces: An Efficient New Technique for State-Space Reduction*. In Helmut Seidl, editor: Programming Languages and Systems, Lecture Notes in Computer Science 7211, Springer Berlin Heidelberg, p. 274–294, doi:10.1007/978-3-642-28869-2_14. arXiv:1204.0414.

[6] Lisbeth Fajstrup, Éric Goubault, Emmanuel Haucourt, Samuel Mimram & Martin Raussen (2016): *Directed Algebraic Topology and Concurrency*. Springer International Publishing, doi:10.1007/978-3-319-15398-8.

[7] Lisbeth Fajstrup, Martin Raußen & Éric Goubault (2006): *Algebraic topology and concurrency*. Theoretical Computer Science 357(1-3), pp. 241–278, doi:10.1016/j.tcs.2006.03.022.

[8] Jean-Yves Girard (2001): *Locus solum: From the rules of logic to the logic of rules*. Mathematical structures in computer science 11(3), p. 301, doi:10.1017/S096012950100336X.

[9] Patrice Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems*. Lecture Notes in Computer Science 1, Springer, Berlin, Heidelberg, doi:10.1007/3-540-60761-7.

[10] É. Goubault, T. Heindel & S. Mimram (2013): *A geometric view of partial order reduction*. Electronic Notes in Theoretical Computer Science 298, pp. 179–195, doi:10.1016/j.entcs.2013.09.013.

[11] Éric Goubault (2003): *Some geometric perspectives in concurrency theory*. Homology, Homotopy and Applications 5(2), pp. 95–136, doi:10.4310/HHA.2003.v5.n2.a5.

[12] Éric Goubault & Emmanuel Haucourt (2005): *A practical application of geometric semantics to static analysis of concurrent programs*. In: International Conference on Concurrency Theory, Springer, pp. 503–517, doi:10.1007/11539452_38.

[13] Emmanuel Haucourt (2018): *The geometry of conservative programs*. Mathematical Structures in Computer Science 28(10), p. 1723–1769, doi:10.1017/S0960129517000226.

[14] Samuel Mimram & Aly-Bora Ulusoy (2021): *Sparkling*. Available at `https://smimram.github.io/sparkling/`.

[15] Peter W O'Hearn (2007): *Resources, concurrency, and local reasoning*. Theoretical computer science 375(1-3), pp. 271–307, doi:10.1007/978-3-540-28644-8_4.

[16] Susan Owicki & David Gries (1976): *An axiomatic proof technique for parallel programs I*. Acta informatica 6(4), pp. 319–340, doi:10.1007/978-1-4612-6315-9_12.