A syntactic approach to the directed semantics of concurrent programs

Aly-Bora Ulusoy

October 26, 2023

Not startled, like a lion at sounds
Not snared, like the wind in a net
Not soiled, like a lotus by water
Like the horn of the rhinoceros
Wander alone

Without resistance in all directions
Content with whatever you get
Enduring troubles undismayed
Like the horn of the rhinoceros
Wander alone

However, should you find a companion
Upon whose soul you can rest yours
Across all manner of adversity
Brimming with joy
Wander with him

– Extracts from the *Rhinoceros Sutra*

Acknowledgments

Contents

Introduction (Français)

Vérification de programmes concurrents

Le Dîner des Philosophes

Nous sommes en l'an 2000. Le monde est très différent depuis la révolte robotique de la fin des années 90. Trois philosophes, Hume, Wittgenstein et Russell, ramenés à la vie pour lutter contre la menace mécanique, sont en train de diner. Cependant, l'humanité est en train de perdre la guerre et leurs vivres s'amenuisent.

À leur table, seules trois baguettes ont été placées. Une entre chacune de leurs places. Chacun des philosophes suit un ordre strict pour leur dîner qu'ils ne briseront pour rien au monde :

1. Prendre la baguette à leur droite.
2. Prendre la baguette à leur gauche.
3. Manger leur repas.
4. Reposer la baguette à leur droite.
5. Reposer la baguette à leur gauche.

Si l'un des philosophes arrive à saisir deux baguettes, alors tous les philosophes pourront manger leur diner et reprendre leurs efforts pour sauver l'humanité. Cependant, si tous récupèrent la baguette à leur droite en même temps, ils seront piégés éternellement à attendre que leur voisin respectif libère sa baguette. Têtus comme ils sont, l'humanité finira par périr avant qu'un seul des philosophes ne commence son repas.

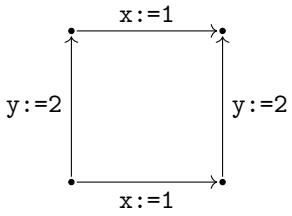
Programmes concurrents

Dans le cas hautement probable qu'une telle situation se produise, nous nous devons d'étudier de tels problèmes. Heureusement, ceux-ci sont liés à au cas plus concret d'accès concurrent aux ressources en programmation parallèle. En effet, le dîner des philosophes est un problème qui fut présenté par Dijkstra dans [12]. Dans cet exemple, chaque philosophe représente un processus en parallèle et les baguettes, une ressource qu'ils peuvent protéger (verrouiller) avant d'effectuer leurs tâches pour s'assurer qu'aucun autre processus n'ait accès à la même ressource au même moment. Lorsque l'accès et l'écriture à la mémoire ne sont pas des opérations atomiques, la modification de la mémoire peut avoir des résultats incontrôlables. D'où l'importance d'utiliser de telles pratiques.

L'utilisation de programmes concurrents est de plus en plus répandue, mais ces programmes sont notoirement plus complexes à concevoir et à vérifier. Les propriétés du

programme (comme l'atteignabilité d'un état) doivent être vérifiées indépendamment de l'ordre relatif dans lequel chaque instruction des différents processus en parallèle est exécutée.

Historiquement, les premiers modèles de la concurrence furent les *modèles d'entrelacements*. Ces modèles sont construits à partir de toutes les différentes exécutions possibles d'un programme. Par exemple, en considérant le programme `x:=1`$\|$`y:=2`, composé de deux instructions d'affectation exécutées en parallèle. Le modèle d'entrelacement associé serait le graphe suivant avec quatre sommets et quatre arêtes.



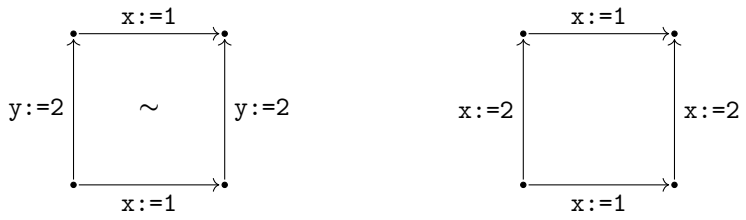
Les deux ordonnancements potentiels des affectations `x:=1` et `y:=2` sont données par les chemins maximaux libellés par `x:=1` $\cdot$ `y:=2` et `y:=2` $\cdot$ `x:=1`. En théorie, il serait possible d'appliquer des techniques de vérifications classiques de programmation séquentielle sur chacun des ordonnancements possibles. En pratique, ceci n'est pas faisable, car le nombre d'entrelacements grandit exponentiellement avec la taille du programme.

La sémantique ci-dessus ne distingue pas si deux actions sont indépendantes ou non. En effet, dans notre cas, les deux exécutions mènent au même état à la fin des instructions `x:=1` et `y:=2`, que nous définissons comme étant indépendantes. Ce ne serait pas le cas si, par exemple, nous remplacions l'instruction `y:=2` par l'instruction `x:=2`. En effet, dans ce cas, la dernière instruction exécutée détermine l'état final de la mémoire

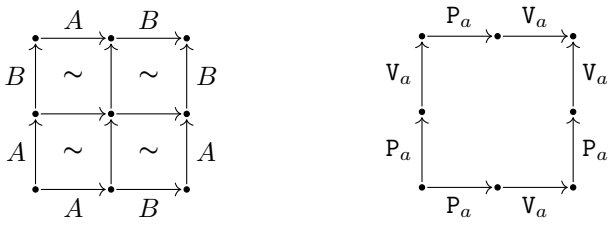
Lorsque deux ordonnancements sont identiques à permutation d'instructions indépendantes près, on dit qu'ils sont équivalents. Dans un tel cas, au lieu de vérifier chaque ordonnancement, il suffit d'en vérifier un seul par classe d'équivalence.

True Concurrency

La remarque du dessus suggère qu'un modèle des programmes concurrent ne devrait pas seulement prendre en compte tous les ordonnancements possibles d'un programme, mais également la commutation d'instructions indépendantes, appelé *True Concurrency*. À fin de distinguer les chemins indépendants de ceux qui ne le sont pas, les graphes sont équipés d'une relation $\sim$ sur les chemins, qui indique leur équivalence. Nous obtenons ainsi, ce qu'on appelle des *graphes asynchrones*. Celui associé à notre exemple précédent est fourni ci-dessous.



La plupart des systèmes d'exploitations, fournissent un cas particulier des ressources abordées précédemment, appelées des mutex, qui ne peuvent être verrouillées par au plus un seul processus à la fois. Étant donné un mutex a, un processus peut soit verrouiller le mutex avec l'instruction $\mathtt{P}_a$, soit le libérer avec l'instruction $\mathtt{V}_a$. À l'instar de nos philosophes attendant leur baguette, un processus essayant de verrouiller un mutex qui l'est déjà sera figé en l'attente de celui-ci. L'utilisation de ces instructions a deux effets sur la sémantique de nos programmes : elle interdit l'accès à certains états de la mémoire (ceux qui correspondent au verrouillage d'un mutex par deux processus distinct simultanément) et par cette action déclare certains chemins comme explicitement non-équivalents. La sémantique des programmes $(A;B)\|(A;B)$ et $(\mathtt{P}_a;\mathtt{V}_a)\|(\mathtt{P}_a;\mathtt{V}_a)$ est respectivement donnée ci-dessous.

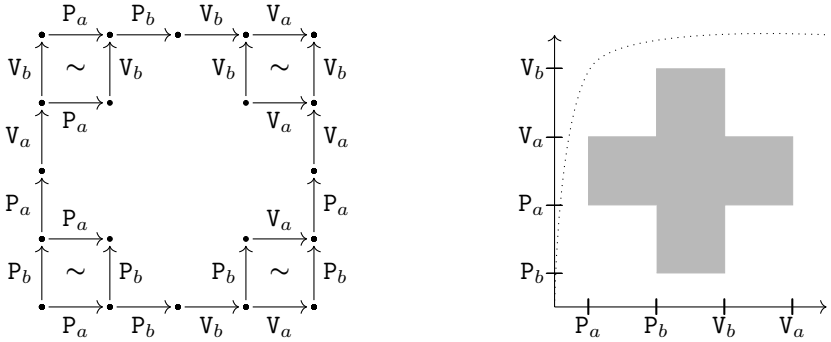


Dans le graphe de gauche, toute paire de chemin du début à la fin du graphe est équivalente, alors que dans le graphe de droite (obtenu en retirant les positions "interdites"), il n'y a plus aucun chemin équivalent. La plupart des modèles de la concurrence que nous présentons dans cette thèse supposeront que la consommation (la différence entre le nombre de fois qu'une ressource donnée est verrouillée et libérée) ne dépend pas du chemin choisit, mais uniquement de ses extrémités. De tels programmes sont appelés *conservatifs*. Ceci peut paraitre restrictif, mais en plus d'être théoriquement très utile, c'est aussi une pratique standard imposée par beaucoup de systèmes concurrents.

Géométrie des programmes

Dans les graphes asynchrones présentés précédemment, l'exécution de programme correspond à des chemins sur les graphes. Les carrés commutatifs peuvent être vus comme des carrés "remplis", qui intuitivement permettent d'effectuer des déformations continues entre les chemins sur les extrémités de ce carré. Ces modèles auraient donc une structure algébrique avec une interprétation géométrique agréable. Ils correspondent à des espaces topologiques dans lesquels l'exécution correspond à des chemins et les équivalences d'instructions correspond à l'homotopie de chemin, c'est-à-dire une équivalence à déformation continue près.

Par exemple, nous pouvons associer au programme $(\mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a)||(\mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b)$, appelé la *Croix Suisse* l'espace topologique ci-dessous, qui correspond au complémentaire de la région grisée dans l'espace $[0,1]\times[0,1]$. Les points de la région grisée correspondent aux positions interdites précédemment mentionnées, qui sont obtenues par double verrouillage d'un mutex.



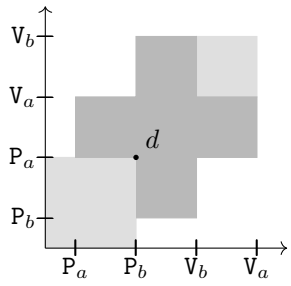
Dans cet espace, les chemins qui commencent dans le coin en bas à gauche et qui sont "croissants", c'est-à-dire ceux qui vont vers le haut et la droite, correspondent aux exécutions. Par exemple, le chemin en pointillés correspond à l'ordonnancement où toutes les instructions du premier processus sont exécutées en premier, puis toutes celles du deuxième processus.

Les chemins qui ne sont pas croissants ne correspondent pas à l'exécution d'un programme, qui ne peut se faire que dans une direction, celle du temps. C'est pourquoi nous avons besoin de considérer une variante *dirigée* des espaces topologiques, équipés de structure supplémentaire qui spécifie la direction du temps et quels chemins sont croissants.

On peut donc étudier une notion d'homotopie dirigée, qui doit une fois de plus correspondre aux classes d'équivalence d'exécution. Le sémantique topologique dirigée peut être reliée à nos graphes asynchrones, en effet en comparant notre exemple précédent, il semble y avoir un lien fort entre les deux.

Vérification de programmes

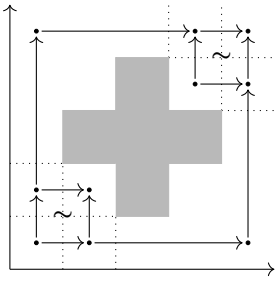
Cette connexion entre la topologie algébrique et la sémantique de programmes concurrents nous fournit un nouveau point de vue sur ces programmes et permet de formuler de nouveaux algorithmes pour leur vérification.



Dans la sémantique ci-dessus, le point d est un *deadlock*. De ce point, il n'y a aucun chemin strictement croissant, c'est-à-dire qu'aucune instruction ne peut être exécutée. Ce problème est spécifique aux programmes concurrents et arrive lorsque des processus (ou des philosophes) attendent mutuellement que l'autre libère une ressource (ou une baguette) Tous les points qui peuvent atteindre un deadlock par un chemin "dirigé" sont appelés *à risque*. Ils sont représentés par le carré en bas à gauche. Des points qui ne

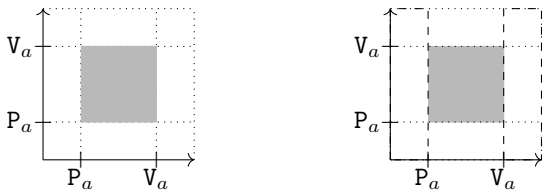
peuvent qu'atteindre un deadlock sont appelés *condamnés*. Les points du carré en haut à droite sont *inatteignables*. Ils ne peuvent pas être atteint depuis un chemin dirigé partant du début de notre exécution. Ces points ne sont pas aussi problématiques que les précédents, mais peuvent révéler des défauts de conception dans le code (dont toutes les instructions devraient avoir une utilité).

Une autre application des techniques géométriques est la réduction du nombre de chemins et d'états à explorer. Ces méthodes sont basées sur l'idée que des chemins équivalents correspondent à des chemins homotopes. De la sémantique topologique dirigée d'un programme, une description compacte peut être obtenue par la *catégorie de composantes* qui identifie les parties du programme où "rien ne se passe" du point de vue de la concurrence. Reprenons l'exemple de la Croix Suisse. Sa catégorie de composante est donnée plus bas, où la commutation des diagrammes est signalée par $\sim$.



Représentation efficace des programmes

Un des problèmes des modèles topologiques est que l'espace d'états est infini, et donc difficilement implémentable. Ceci a motivé la recherche d'une représentation efficace des espaces topologiques dirigés. Dans le cas de la composition parallèle de n processus séquentiels (sans branchements, ni boucles), appelé un *programme simple*, la sémantique géométrique associée est une version dirigée du n-cube standard $[0,1]^n$, duquel un nombre fini de cube de dimension n ont été retirés. Dans cette situation, il est possible de recouvrir l'espace par un ensemble fini de n-cubes, que l'on appelle un recouvrement cubique. Un espace admettant un tel recouvrement est appelé une région cubique. Ces régions sont d'un intérêt particulier, car les régions interdites et leur complémentaire sont des régions cubiques.



Nous remarquons facilement que toutes les représentations d'un même espace $X \subseteq [0,1]^n$ par un recouvrement cubique ne sont pas d'une qualité égale. En effet, dans les recouvrements de l'exemple ci-dessus, le recouvrement de gauche, composé de 8 est moins efficace que celui de droite qui ne contient que 4 cubes. Nous verrons qu'il existe en effet une notion naturelle de "meilleure" représentation cubique pour un espace donné, qui corre-

spond à tous les n-cubes maximaux par la taille qu'il contient. Dans l'exemple ci-dessus, cela correspond au recouvrement sur la droite.

Cette représentation est le fondement des algorithmes de détection de deadlocks existant sur les modèles géométriques et contiennent une grande quantité d'information sur l'indépendance des instructions, qui permet de définir la catégorie de composante susmentionnée à partir de la donnée du recouvrement maximal.

Factorisation des programmes concurrents

Une autre approche pour réduire l'espace d'états d'un programme simple est de le séparer en plusieurs groupes de processus indépendants. Des processus sont dit indépendants quand n'importe laquelle de leurs instructions peuvent être permutées dans un ordonnancement sans effet sur l'état du programme.

Ceci implique que pour deux processus P_1 et P_2, indépendants selon la définition ci-dessus, la sémantique géométrique de leur composition parallèle sera obtenue comme le produit de la sémantique de P_1 et de celle de P_2. Cette propriété s'étend également à la catégorie de composantes et d'autres invariants dirigés de nos programmes. En un sens, cette indépendance est une généralisation de la notion d'indépendance des instructions présentée précédemment. Nous pourrons remarquer que l'existence d'une telle factorisation permet de décrire l'espace d'états de façon beaucoup plus compacte.

Contributions

Programmes concurrents et le théorème d'Hashimoto

Dans son papier [26], Hashimoto prouve un puissant théorème sur les ordres partiels dont tous les éléments sont connectés par des zigzags de comparaison. En effet, pour de tels ordres partiels, toute paire isomorphe de décomposition sous forme de produit admet une décomposition "plus fine".

Plus précisément, dans un ordre partiel connecté, pour toute paire $\prod_{\alpha\in A} X_\alpha$ et $\prod_{\beta\in B} Y_\beta$ de décomposition en produit, il existe une décomposition $\prod_{\alpha\in A}\prod_{\beta\in B} Z_{\alpha,\beta}$ telle que $X_\alpha \cong \prod_{b\in B} Z_{\alpha,b}$ et $Y_\beta \cong \prod_{a\in A} Z_{a,\beta}$ pour tout α, β. Cette propriété est appelée *propriété de raffinement forte* et implique, entre autre que pour tout ordre partiel fini, il existe une unique décomposition, à isomorphisme près, sous forme de produit d'ordres partiels irréductibles. Ce résultat est cependant beaucoup plus fort car la propriété de raffinement existe également pour des produits infinis.

Considérons l'espace euclidien $\mathbb{R}^3$ se décomposant des deux façons suivantes $\mathbb{R}^2 \times \mathbb{R}$ et $\mathbb{R} \times \mathbb{R}^2$. Nous avons le raffinement suivant $\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R} \times \mathbb{R} \times \{*\} \times \mathbb{R}$ avec les affectations suivantes:

$$Z_{1,1} = \mathbb{R} \qquad Z_{1,2} = \mathbb{R} \qquad Z_{2,1} = \{*\} \qquad Z_{2,2} = \mathbb{R}$$

Dans le chapitre 2, nous élargissons ce résultat pour le cas des catégories connectées *sans boucles*, qui sont des catégories sans endomorphismes non triviaux. Les catégories sans-boucles sont des généralisations directes des ensembles partiellement ordonnés, qui vérifient la même propriété (une sorte d'anti-symétrie généralisée). Contrairement aux ordres partiels, les catégories sans boucles peuvent admettre plusieurs morphismes en parallèles, les rendant plus complexes à étudier. Beaucoup des invariants mentionnés

précédemment, comme la catégorie de composante, sont des catégories sans boucle lorsque le programme est lui-même sans boucle, ce qui motive notre approche.

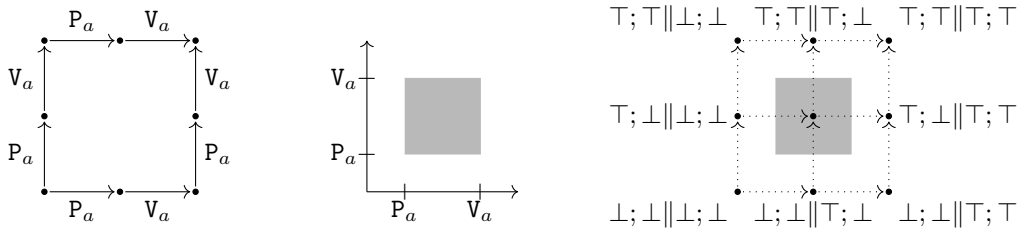
L'existence d'une décomposition sous forme de produit de cette catégorie non seulement réduit la taille des catégories à étudier, et donc de notre représentation, mais donne également une factorisation de notre programme comme parallélisation de processus indépendants.

Approche syntactique des programmes

Les modèles topologiques permettent l'utilisation d'outils et de résultats théoriques très puissants. Cependant, ceux-ci n'amènent pas toujours aux algorithmes les plus simples à implémenter. Nos travaux commencent par l'observation que pour beaucoup des outils utilisés dans les sémantiques topologiques, très peu découlent des propriétés topologiques de nos espaces, mais de la syntaxe des programmes. Dans le chapitre 3, nous définissons un nouveau modèle de programmes, représenté par un ordre partiel induit par des graphes dirigés. Au lieu de construire l'espace des états de façon globale, nous fournissons des constructions inductives pour les états du programme qui permettent de représenter plus efficacement l'espace d'états. Cela nous donne une représentation très intuitive des positions de nos ordres partiels comme des préfixes d'une exécution.

Pour une unique instruction, nous ne considérons que deux états : $\bot$ (l'instruction n'a pas été exécutée) et $\top$ (l'instruction a été exécutée). Pour des constructions plus complexes, par exemple la composition séquentielle $P;Q$, nous définissons inductivement deux ensembles d'états possibles : $p;\bot$, où p est un état de P et $\top;q$, où q est un état de Q. Pour la parallélisation $P\|Q$, les états sont définis par des paires d'états $p\|q$.

Nous obtenons des modèles très proches des graphes asynchrones, à la différence que nous ne retirons pas les positions interdites, mais que nous vérifions si celles-ci font parties des cubes que nous considérons. Même si cela ne parait pas très efficace, c'est ce qui nous permet de retrouver les constructions des modèles géométriques avec une implémentation efficace.



Ensuite nous réimplémenterons la représentation compacte des régions autorisées et interdites des programmes par des ensembles de cubes. Dans notre modèle, ces cubes sont naturellement définis par des paires de positions (p, q) de nos programmes qui recouvrent des régions, que l'on appelle leur support, qui contient toutes les positions x telles que $p \leq x \leq q$. Nous obtenons des résultats similaires quant à l'existence d'une "meilleure représentation" des régions par une couverture maximale dans le cas des programmes conservatifs. Nous obtenons également un algorithme de détection de deadlock utilisant les positions et cubes syntaxiques, dont une implémentation est donnée sur notre site internet[1]. Notre implémentation supporte un grand nombre de programme conservatifs,

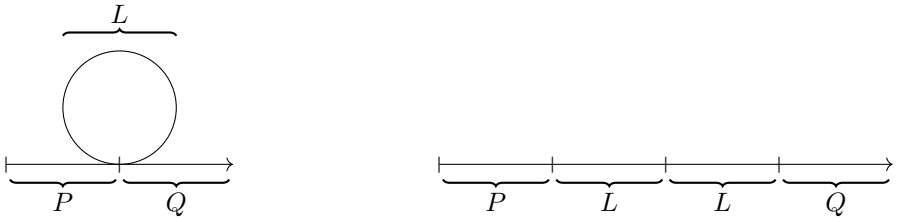
[1] `https://smimram.github.io/sparkling/`

dont une petite librairie est donnée en exemple pour que le lecteur curieux puisse s'y essayer. Notre implémentation, par la nature inductive de nos positions est très efficace et plus intuitive que l'approche géométrique.

Vérification des programmes avec des boucles

La vérification des programmes avec des boucles est d'une complexité notoire et les programmes concurrents ne font pas exception à la règle.

Nombre des outils présentés dans le chapitre 1 échouent en présence de boucles. Dans le chapitre 4, nous allons élargir notre modèle syntaxique afin de pouvoir appliquer les algorithmes de détection de deadlock dans le cas de programmes avec des boucles. L'idée derrière cette méthode est assez simple. Chaque boucle est remplacée par deux copies du code qu'elle contient, composées de façon séquentielle.



Une piste similaire a déjà été explorée sur un algorithme de détection différent dans le papier [17]. Cependant, la borne supérieure du nombre de dépliages nécessaire pour obtenir la région à risque et condamnée est élevée.

Nous prouverons, qu'avec notre méthode, il suffit de déplier chaque boucle deux fois afin d'obtenir les régions concernées. Nous obtenons ce résultat en élargissant la correspondance entre les points du dépliage et du programme de base aux recouvrements cubiques de ceux-ci. Cette approche est bien plus compliquée, puisqu'elle implique d'élargir la notion d'intervalle et de cube aux préordres.

Introduction

Verification of concurrent programs

Dining Philosophers

It is the distant future, the year 2000. The world is vastly different since the robotic uprising of the mid-nineties. Three philosophers, Hume, Wittgenstein and Russell, brought back to life to fight against the machine threat, are having dinner. But the forces of humanity are losing the war and supplies are running low. At their table, only three chopsticks are set. One between each seat. The philosophers each follow a very strict order for their dinner, that they shall never break:

1. They take the chopstick to their right.
2. They take the chopstick to their left.
3. They eat their meal.
4. They put back the chopstick to their right
5. They put back the chopstick to their left.

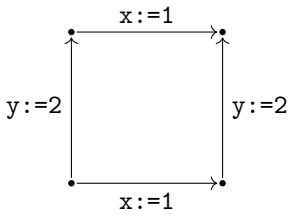
Should one of the philosophers get access to both chopsticks, then all the philosophers can have their meal and resume their quest to free mankind. However, should they all get a hold of the chopstick on their right at the same time, they will all be locked in a deadly contest, each waiting on each other to release the fateful cutlery on their left before doing anything else. Stubborn as they are, humanity shall perish at the hand of its unfeeling oppressor before a single philosopher has started its meal.

Concurrent programs

In case of such a highly probable situation ever occurring, we must study these problems, which are thankfully linked to the more concrete problem of concurrent access to resources in parallel programming. Indeed, the dining philosophers was an example introduced by Dijkstra in [12] to represent multiple processes running in parallel. In this example each of our philosopher represent a process and the chopsticks, a resource, that they can protect (lock) before performing tasks to ensure that no other process access the same resource at the same time. This is important as in most models of memory, where access and writing to the memory is not an atomic operation, concurrent modification of the memory can have undefined results (two processes incrementing a variable by 1 at the same time might increase it by only 1 instead of 2).

The use of concurrent programs has become more and more widespread in order to exploit the features of more recent architectures (multicore processors, clouds, etc.), but they are notoriously complex to design and conceptualize. Properties of the program (such as not reaching a certain memory state) have to hold regardless of the relative order in which each instruction of the different processes constituting the program are executed.

Historically, the first models of concurrent programs were the *interleaving models*. These models essentially consist of all different possible executions of a program. For instance, considering the program `x:=1`$\|$`y:=2`, consisting of two affectation instruction executed in parallel, the associated interleaving model would be the following graph with four vertices and four edges.



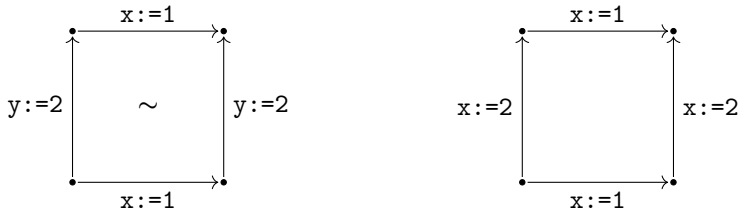
The two possible interleaving of the affectations `x:=1` and `y:=2` are given by the maximal paths labelled by `x:=1` $\cdot$ `y:=2` and `y:=2` $\cdot$ `x:=1`. In theory, we could apply traditional verification techniques for sequential program on each potential ordering. This is not feasible in practice because of the sheer number of these executions, which may grow exponentially with the size of the program.

One could remark that the semantics above do not distinguish whether two actions are independent or not. Indeed, in our case both executions lead to the same state at the end of `x:=1` and `y:=2`, which we define as being independent. This would not be the case if, for example we replace the instruction `y:=2` with the instruction `x:=2`. Indeed, in that case, whichever instruction is executed last determines the final state of the memory.

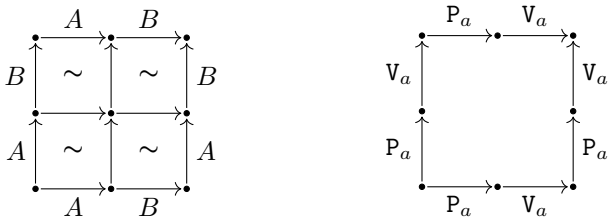
When two schedulings are the same up to permutation of independent instructions, we say that they are *equivalent*. Then, instead of checking all the potential executions, one can simply check a single execution in each equivalence class to verify the properties of the program.

True concurrency

This suggests that a model for concurrent programs should not only incorporate all possible schedulings of a program (as in traditional interleaving semantics), but also the commutation of independent instructions, following the principle of what is now called *true concurrency*. In order to distinguish between paths that are independent and those that are not, the graphs are equipped with a relation $\sim$ on paths, indicating whether they are equivalent in the way explained above. We thus obtain what are called asynchronous graphs, given for our previous examples below:



Most operating systems offer a particular kind of resource, called a mutex, which can be held by at most one process at a time: given a mutex a, a process has either the option of locking or releasing the mutex by respectively performing the instructions $\mathtt{P}_a$ or $\mathtt{V}_a$. Like our philosophers waiting on their chopsticks, a process trying to lock a mutex held by another process will remain frozen until that resource is released. The usage of these instructions has two effects on the semantics: it forbids some states (any states that corresponds to two different process locking the same resource a with the instruction $\mathtt{P}_a$ has to be removed) and through this process explicitly declares some paths as not equivalent. For instance, the semantics of $(A;B)\|(A;B)$ and $(\mathtt{P}_a;\mathtt{V}_a)\|(\mathtt{P}_a;\mathtt{V}_a)$ are respectively:



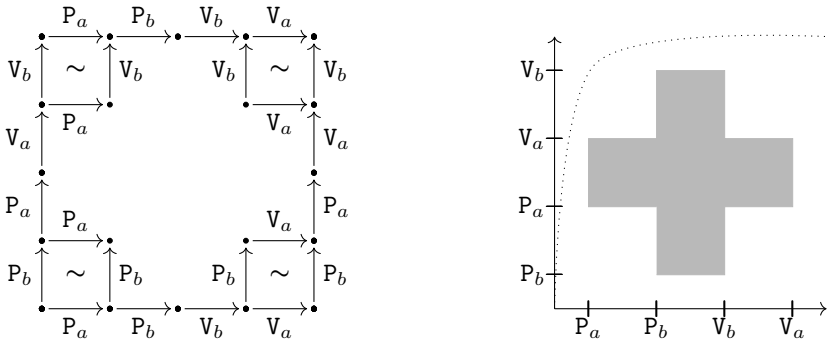
In the first graph, any two paths from start to finish are equivalent, while in the second graph, (which can be obtained by removing the "forbidden" central position and related vertices), there are no equivalent paths. Most of the models of concurrency we will present assume that two paths on our state space reaching the same point will have locked/released a given resource the same number of times. Meaning that the "consumption" of resources does not depend on the path but only the endpoints. We say that such programs are *conservative*. This is a very useful consideration in our study of concurrent programs that is also generally enforced in practice.

Geometry of programs

In the asynchronous graph semantics presented above, the executions of the program correspond to paths in the graph. Moreover, squares filled with $\sim$ can be seen as "filled squares" in which intuitively, there is room to allow for a deformation between the paths on the extremities of this square. This means that the resulting models bear algebraic structure with a nice geometric interpretation: roughly, they correspond to topological spaces, in which executions correspond to paths and the equivalence between two executions corresponds to homotopy of paths, i.e. equivalence up to continuous deformation.

For instance, to the program $(\mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a)||(\mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b)$, called the *Swiss Cross*, is associated the topological space on the right below, which correspond to the complement

of the darkened region in the space $[0,1]\times[0,1]$. Points in the darkened regions correspond to the forbidden states mentioned above, which correspond to mutexes being locked twice.



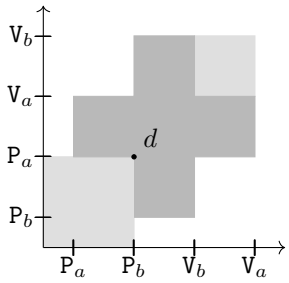
In this space, paths that start from the lower-left corner and which are "increasing", i.e. they only go up and to the right, correspond to executions. As an example, the dotted path traced above, will correspond to the scheduling where all instructions of the first process are executed first followed by all instructions of the second process.

Paths which are not increasing do not make sense from a computational point of view, they cannot correspond to the execution of a program. Indeed, going in any other direction than the direction of time would be equivalent to "un-executing" an instruction. That is why to study program one needs to consider a *directed* variant of topological spaces, which comes equipped with a notion of direction of time and extra structure, that specifies which paths are directed.

One can then study the geometry of these spaces. In particular, the structure of these directed paths up to a suitable notion of homotopy, which should once again correspond to equivalence classes of executions, up to commutation of independent actions. The directed topological semantics can also be related to the asynchronous semantics. Indeed, if we compare our space to the asynchronous graph, there does seem to be a strong link between the two.

Verification of programs

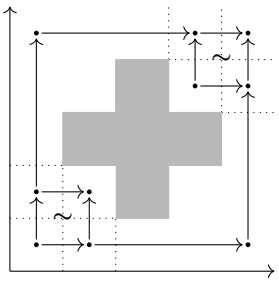
This connection between algebraic topology and the semantics of concurrent program provides with a new point of view on those programs and allows for the formulation of new algorithms for their verification.



In the semantics above, the point d is a *deadlock* point. Starting from this point, there is no strictly increasing path, i.e. no instructions can be executed. This kind of undesirable

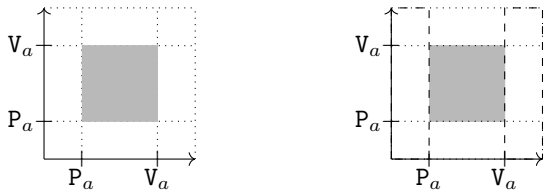
behaviour is specific to concurrent programs, and occurs when processes (or philosophers) are waiting on each other to free up a resource (the chopsticks). All points that can reach a deadlock by a "directed" path, are called *unsafe*. They are represented in the lower left corner. They are states from which an execution might lead to a deadlock. If the only points that can be eventually reached are deadlocks, then the states are called *doomed*. Points, in the upper right corner, are called *unreachable*, they cannot be reached by a directed path from the initial state of the program, i.e. they are states that will never appear in an execution. While this is not a concern in itself, it reflects poor design in the programming (or worse). Based on the geometric characterization of these states, we can derive algorithms to compute them, providing guarantees about the programs' safety.

Another application of the geometric techniques is to reduce the number of paths and states to explore based on the idea that equivalent paths (in the sense of permutation of independent actions) correspond to homotopic paths. From the directed topological semantics of a program, a compact description can be obtained by the *category of components* which identifies portions of the program where "nothing happens" from the concurrent point of view. This is done be reducing these paths to identities. As an example, consider the Swiss Cross from above. We give below its category of components, where commutation of actions is expressed by $\sim$. In any of the regions delimited by dotted lines, any path we take does not change how we could've reached that state or the states we can reach later on.



Efficient representation of programs

One problem with our topological models is that the state space is infinite and thus difficult to implement. This, amongst other things, has motivated the quest for an efficient representation of the directed topological spaces that we are dealing with. In the case of parallel composition of n sequential processes, i.e. processes with no conditional branching or loops, called *simple programs*, the geometric semantics associated correspond to a directed version of the standard n-cube $[0,1]^n$, which consists of the product of n copies of the unit interval, with a finite amount of cubes that have been removed. In this situation we can cover the space with a finite set of n-cube (product of n intervals), called a cubical cover as shown in the example below. When a space admits a cubical cover, we say it is a cubical region, which will be of special interest as forbidden positions and their complement do correspond to such regions.



It is easy to see that there are better ways to represent a specific space $X \subseteq [0,1]^n$ by a cover of n-cube. Indeed, in the example above, the cover on the left with 8 cubes is less efficient, as it uses more intervals to talk about the same space as the cover on the right (which only has 4 cubes). We will see that there is indeed a natural notion of maximal cover for a given space, corresponding to all the maximal intervals that cover it. In the example above, it does indeed correspond to the cover given on the right.

This representation is the basis of the existing deadlock detection algorithm implemented on these geometric models, and also encodes much information about independence of instructions which allows one to define the category of components of the geometric semantics of a simple program from the maximal cover.

Factorization of concurrent programs

Another way of reducing the size of the state space of a simple program, is to split it as several groups of processes running independently of each other. We say that to processes are independent when any of their instructions can be permuted in any scheduling without any effect of the state. What this implies is that for two processes P_1 and P_2, independent in the sense above, the geometric semantics of their parallel composition $P_1 \| P_2$ can be obtained as the product of the geometric semantics of P_1 and the geometric semantics of P_2. This also extends to the category of components and other invariants of our programs. In a sense, this is a generalization of the notion of independence of instructions presented above. One should note that the existence of such a *factorization* allows one to describe the state space in a much more compact way. This process introduced in [31], is very similar to the factorization of integers as products of prime factors or of polynomials as products of monomials, and rely on the maximal covers introduced above.

Contributions

Hashimoto's theorem and concurrent programs

In [26], Hashimoto has proved a powerful theorem, stating that, for partial orders where all elements are connected by a "zigzag" of comparisons, which we call connected posets, for any pair of isomorphic product decomposition, there exists a "finer" decomposition.

More precisely, in a connected poset, for any pair of product decompositions $\prod_{\alpha \in A} X_\alpha$ and $\prod_{\beta \in B} Y_\beta$ that are isomorphic, there exists a decomposition $\prod_{\alpha \in A} \prod_{\beta \in B} Z_{\alpha,\beta}$ such that $X_\alpha \cong \prod_{b \in B} Z_{\alpha,b}$ and $Y_\beta \cong \prod_{a \in A} Z_{a,\beta}$ for any α, β. This is called the *strict refinement property*, and implies, among other things, that for any finite connected poset, there exists a unique decomposition as a product of irreducible factors (up to isomorphism). But this result is much more powerful than this, as it also gives a "better" factorization for any pair of infinite products.

For example, if we look at the Euclidean space $\mathbb{R}^3$, for the decompositions $\mathbb{R}^2 \times \mathbb{R}$ and $\mathbb{R} \times \mathbb{R}^2$, we have a refinement $\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R} \times \mathbb{R} \times \{*\} \times \mathbb{R}$ gives a refinement of these two decompositions by assigning:

$$Z_{1,1} = \mathbb{R} \qquad Z_{1,2} = \mathbb{R} \qquad Z_{2,1} = \{*\} \qquad Z_{2,2} = \mathbb{R}$$

In Chapter 2, we extend this result to the case of connected *loop-free* categories, which are categories with no non-trivial endomorphisms. Loop-free categories are a natural generalization of posets, which when seen as categories verify the same condition (which can be seen as more general notion of antisymmetry). But contrary to posets, loop-free categories might admit multiple parallel morphisms. These categories are a natural setting in which most of the topological invariants used in practice naturally appear. One very important example is the category of components mentioned earlier, which gives a compact representation of our programs.

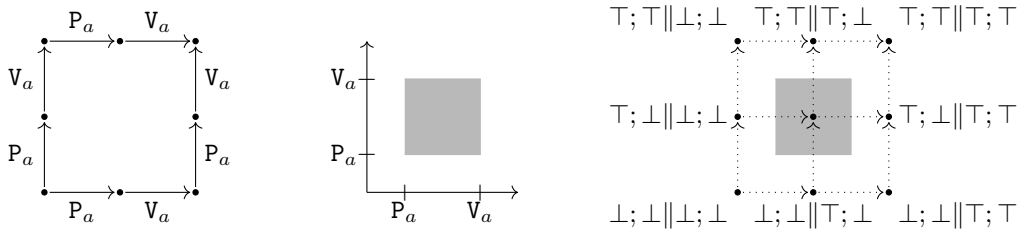
The existence of a product decomposition for this category not only naturally reduces the size of the categories to study, and thus of our representation of the state space, but will also give a factorization of our program as a composition of independent processes (whose actions can be freely permuted in the scheduling of an execution).

A syntactic approach to programs

Although topological models bring with them quite powerful tools and theoretical results, the resulting algorithms they lead to are notoriously hard to implement. Our work starts by observing that many of the tools that are actually used in the directed topological models (cubical covers, category of components...) do not stem from the topology associated but from the underlying, discrete syntax of the program. Most constructions are consequences of the directed nature of this syntax, and the very restrictive structure of programs. In Chapter 3 we define a new model of programs, represented by partial orders induced by directed graphs, but instead of constructing the state space globally, we provide an inductive construction for the states themselves, which allows us to represent more efficiently the state space. This gives us a very natural interpretation of positions in our posets as prefixes of executions.

For a simple instruction, we consider only two states $\bot$ (not executed) and $\top$ (already executed). For more complex constructions, for example sequential composition $P;Q$, we define inductively the possible states: $p;\bot$, where p is a state of P and $\top;q$, with q a state of Q. For parallel composition $P\|Q$, the states are defined as pairs of states $p\|q$.

We obtain models very close to asynchronous graphs, excepting that we do not remove the forbidden positions, but check if our cubes intersect with the set of forbidden cubes. This might appear less efficient, but not removing these positions is what allows us to get back the tools from directed topology efficiently, as the resulting sets of positions are extremely simple and intersection is not costly. Below, we can compare from left to right the asynchronous graph, directed geometric semantics and syntactic semantics of the program $\mathtt{P}_a;\mathtt{V}_a\|\mathtt{P}_a;\mathtt{V}_a$.

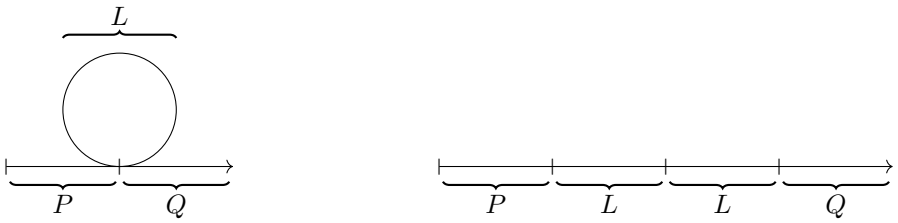


Then, we re-implement the efficient representation of programs as sets of cubes, that are naturally defined by pairs of positions (p, q) in our programs covering regions, that we call their supports comprising all points x such that $p \leq x \leq q$. We obtain similar result for the existence of "best representation" of regions by a maximal cover in the case of conservative programs. We also obtain a similar deadlock detection algorithm using syntactic positions and cover, of which we give an implementation on our website[2] which handles all conservative programs, with a small library of examples for the curious reader to try out. By the inductive nature of our syntactic states, we get a very efficient implementation of the verification algorithms of the geometric approach in the context of our syntactic model.

Verification of program with loops

The verification of programs with loops is notoriously hard, and concurrency problems are no exception to this rule.

Many tools and algorithms that we will present in Chapter 1 fail in the presence of loops. In Chapter 4 we upgrade our new syntactic model to extend the deadlock detection algorithm that we adapted in Chapter 3. The idea behind the extension is quite simple. We replace each loop of our program by two copies of the code inside the loop, composed sequentially.



A similar idea has already been explored on a different algorithm in [17]. But the upper bound on the number of time a single loop must be unfolded in order to find the correct unsafe region is quite high.

We will prove that, with our method, all loops must be unfolded at most twice to recover the correct unsafe and doomed region of our program with loops. We obtain this by not only defining a correspondence between points of the state space of a program and its unfolding, but also proving that this correspondence extends to the cubical covers of these regions. This is much harder to do, as it requires extending the notion of cube/interval to states with loops.

[2] `https://smimram.github.io/sparkling/`

Notations

We follow standard notation and write $\mathbb{N}$ for the set of positive integers, $\mathbb{Z}$ for the set of all integers and $\mathbb{R}$ for the set of real numbers. We write $\mathfrak{P}(X)$ for the powerset of a set X; $Y^{\mathrm{c}} = X \setminus Y$ for the complement of any subset Y of X. We write $[1\colon n]$ for the set $\{1, \ldots, n\}$; and $]x, y[$ (resp. [x,y]) for open (resp. closed) intervals (Following French convention). All the classical concepts and definitions used throughout the thesis can be found in Appendix A

Chapter 1
Directed topological models of concurrency

The story so far: In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.

– Douglas Adams, *The Restaurant at the End of the Universe*

In this chapter, we introduce models based on the trace space (the set of all possible executions and their effect on the memory) and verification techniques for concurrent programming languages. We introduce these models on the **PIMP** language, introduced in [18] which is a slightly modified a version of the **IMP** language [54], equipped to handle threads running in parallel. This language was chosen as a good compromise between more theoretical approaches (CCS, π-calculus [40]) and real world implementations (e.g. Java, POSIX) of parallel computing. After a brief introduction of the language and verification of programs (Section 1.1) we introduce a first simple model of programs (Section 1.2) and adapt it to deal with the specific problems of concurrent programming. Finally, we introduce a more complex model of programs from [18, Chapter 4] based on directed topology in Section 1.3 in order to introduce powerful algorithms based on topological tools to verify programs. In later chapters, we will present our new model of programs, aiming to conserve the powerful tools introduced in this last topological model. All the material of this chapter is heavily sourced and inspired from the presentation in [18].

1.1 Concurrent programming languages

1.1.1 A toy language for concurrent programs

Definition 1.1.1. Let Var be a fixed, countable set of variables. We define the language **PIMP** from four kinds of syntactic expressions, defined by their grammar:

- The set $\mathcal{A}$ of arithmetic expressions:

$$a \quad ::= \quad x \quad | \quad n \quad | \quad a \texttt{ + } a \quad | \quad a \texttt{ * } a$$

  where x is a variable in Var and $n \in \mathbb{N}$.

- The set $\mathcal{B}$ of boolean expressions, or conditions:

$$b \quad ::= \quad \texttt{true} \;\mid\; \texttt{false} \;\mid\; a < a \;\mid\; b \;\texttt{and}\; b \;\mid\; \neg b \;\mid\; \texttt{skip}$$

- The set $\mathcal{X}$ of actions:

$$c_{act} \quad ::= \quad x := a \;\mid\; \texttt{skip}$$

- The set $\mathcal{C}$ of commands, or programs:

$$c \quad ::= \quad c;c \;\mid\; \texttt{while}\ b\ \texttt{do}\ c \quad \mid \quad \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c \;\mid\; c \| c \;\mid\; c_{act}$$

A program is executed with regard to an environment (a certain state of the memory) consisting of the values of each variable, in other terms a function $\sigma\colon \mathsf{Var} \to \mathbb{Z}$. In **PIMP**, variable values are restricted to integers for the sake of simplicity. Arithmetic (resp. boolean) expressions evaluate to integers (resp. boolean) in the usual way. Programs that do not contain the constructor $\|$ are called *sequential program*. We refer to standard textbooks [54] for details about these.

Remark 1.1.2. For any arithmetic (resp. boolean) function symbol $f\colon \mathcal{A}^k \to \mathcal{A}$ (respectively $g\colon \mathcal{B}^k \to \mathcal{B}$) of arity k, we could extend the grammar of $\mathcal{X}$ (resp. $\mathcal{B}$) with the constructors $f(a, \dots, a)$ (resp. $g(b, \dots, b)$). The rules we have given can also be seen as a special case of this definition.

The actions have an effect on the memory of the program and commands define the control flow (the structure) of the program. Their meaning is given in Section 1.1.1

`x:=`a	assign the evaluation of a to the variable x
`skip`	do nothing
c_1`;`c_2	sequentially execute c_1 and then c_2
`if` b `then` c_1 `else` c_2	evaluate the boolean expression b and execute c_1 (resp. c_2) if the result is true (resp. false)
`while` b `do` c	execute the command c as long as the boolean expression b evaluates to `true`
$c_1 \| c_2$	execute the command c_1 in parallel with the command c_2

We will focus our study on concurrent programs, in which subprograms (also called *threads* or *processes*) run in parallel. These programs are used to exploit shared computing resources to their full potential, to take advantage of distributed architecture or to be more reactive to external events. This structure is the most prevalent in parallel computing and can be used to represent most parallel tasks.

Example 1.1.3. A simple instance of a parallel program would be distributed computing where a task (for example processing an image) is first separated by a sequence of instructions p_i into multiple smaller tasks $p_1, \dots, p_k$ (processing a small part of the image) and then assembled back together by a sequence p_f. This program would look like

$$p_i; (p_1 \| \dots \| p_k); p_f$$

Remark 1.1.4. In this thesis, we will suppose that the parallel execution of process is sequentially consistent. That is that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations

of each individual processor appear in this sequence in the order specified by its program", as phrased in [35]. This is not the case for real multiprocessors with relaxed memory models [50]. We will nonetheless make this assumption for the sake of simplicity, but this approach could be refined to encompass semantics with relaxed memory models, even though most modern compilers help ensure sequentially consistent semantics at a higher level.

Remark 1.1.5. In the rest of this thesis, we only consider programs with finitely many threads. Although real programming languages allow for threads to be defined recursively, this would not cover much more practical cases and would introduce a lot more problems and technical details that we would have to work around. Indeed, the reachability of a state for finitely multithreaded programs is already PSPACE-complete [33], and becomes undecidable with recursive threads [47].

1.1.2 Operational semantics

Now that we have a language and programs, we need to describe in a formal way what it means to execute the program. We do this by describing a *semantics* for the language [54]. As described before, our programs consists mostly of actions, boolean expressions and arithmetic expression. These commands are executed w.r.t. to a state, which corresponds to the state of the memory (variables, etc.) and other resources available to the programs, and affect said state by their execution or return a value depending on it. For example, the execution of an action $x := 1$ would modify the memory cell corresponding to x and replace its value by 1. Similarly, a boolean expression such as $x == 1$ would check the cell where x is stored and return `true` or `false` depending on its value.

Definition 1.1.6. We write $\Sigma = \mathbb{Z}^{\mathsf{Var}}$ for the set of *states*, consisting of functions assigning an integer to each variable. The initial state $\sigma_0 \in \Sigma$ is the constant function equal to 0. The *operational semantics* of our programming language consists of three functions:

- $[\![-]\!]_{\mathcal{A}} \colon \mathcal{A} \to (\Sigma \to \mathbb{Z})$ describing the evaluation of arithmetic expressions,
- $[\![-]\!]_{\mathcal{B}} \colon \mathcal{B} \to (\Sigma \to \mathbb{B})$ describing the evaluation of boolean expressions,
- $[\![-]\!]_{\mathcal{X}} \colon \mathcal{X} \to (\Sigma \to \Sigma)$ describing the effect of actions on the state.

As well as a *reduction relation* $\to$ on pairs $\langle \sigma, c \rangle$ consisting of a command $c \in \mathcal{C}$ and a state $\sigma \in \Sigma$, which formally describes how a command evaluates in a given environment.

Thus, for instance, $[\![-]\!]_{\mathcal{A}}$ sends each arithmetic expression to a function from the set of states Σ to the set of integers (corresponding to the evaluation of the expression where the variables of the program have the values given in the state). Given $a \in \mathcal{A}$, we thus write $[\![a]\!]_{\mathcal{A}} \colon \Sigma \to \mathbb{Z}$ for its semantic interpretation and similarly for the other functions.

Definition 1.1.7. Let $x \in \mathsf{Var}, n \in \mathbb{N}$, f (resp. g) an arithmetic (resp. boolean) function of arity k. Given a state $\sigma \in \Sigma$. The *evaluation* of arithmetic expressions is defined as:

$$[\![x]\!]_{\mathcal{A}}(\sigma) = \sigma(x) \quad [\![n]\!]_{\mathcal{A}}(\sigma) = n \quad [\![f(a_1, \ldots, a_k)]\!]_{\mathcal{A}}(\sigma) = f([\![a_1]\!]_{\mathcal{A}}(\sigma), \ldots, [\![a_k]\!]_{\mathcal{A}}(\sigma))$$

The *evaluation* of boolean expressions is defined as:

$$[\![\texttt{true}]\!]_{\mathcal{B}}(\sigma) = \top \quad [\![\texttt{false}]\!]_{\mathcal{B}}(\sigma) = \bot \quad [\![g(b_1, \ldots, b_k)]\!]_{\mathcal{B}}(\sigma) = g([\![b_1]\!]_{\mathcal{B}}(\sigma), \ldots, [\![b_k]\!]_{\mathcal{B}}(\sigma))$$

And the *evaluation* of actions is defined as:

$$[\![\texttt{x := a}]\!]_{\mathcal{X}}(\sigma) = \sigma[x \mapsto [\![a]\!]_{\mathcal{A}}(\sigma)]$$

where $\sigma[x \mapsto n]$ is the function which associates n to x and $\sigma(y)$ to each $y \neq x$.

In the following, when the context is clear we will drop the subscripts for $[\![-]\!]$. The `skip` actions plays a particular role in the reduction.

Definition 1.1.8. Let $x \in \mathsf{Var}, b \in \mathcal{B}, a \in \mathcal{A}$. The *reduction relation* $\to$ is defined inductively by the following inference rules:

$$\frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1\,\texttt{;}\,c_2\rangle \to \langle \sigma', c_1'\,\texttt{;}\,c_2\rangle} \qquad \frac{}{\langle \sigma, \texttt{skip;}c\rangle \to \langle \sigma, c\rangle}$$

$$\frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1 \,\|\, c_2\rangle \to \langle \sigma', c_1' \,\|\, c_2\rangle} \qquad \frac{\langle \sigma, c_2\rangle \to \langle \sigma', c_2'\rangle}{\langle \sigma, c_1 \,\|\, c_2\rangle \to \langle \sigma', c_1 \,\|\, c_2'\rangle}$$

$$\frac{}{\langle \sigma, \texttt{skip} \,\|\, c\rangle \to \langle \sigma, c\rangle} \qquad \frac{}{\langle \sigma, c \,\|\, \texttt{skip}\rangle \to \langle \sigma, c\rangle}$$

$$\frac{[\![b]\!](\sigma) = \top}{\langle \sigma, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\rangle \to \langle \sigma, c_1\rangle} \qquad \frac{[\![b]\!](\sigma) = \bot}{\langle \sigma, \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2\rangle \to \langle \sigma, c_2\rangle}$$

$$\frac{[\![b]\!](\sigma) = \top}{\langle \sigma, \texttt{while } b \texttt{ do } c\ \rangle \to \langle \sigma, c\texttt{;while } b \texttt{ do } c\rangle} \qquad \frac{[\![b]\!](\sigma) = \bot}{\langle \sigma, \texttt{if } b \texttt{ then } c\rangle \to \langle \sigma, \texttt{skip}\rangle}$$

$$\frac{c \in \mathcal{X} \setminus \{\texttt{skip}\}}{\langle \sigma, c\rangle \to \langle [\![c]\!](\sigma), \texttt{skip}\rangle}$$

We write $\to^*$ for the reflexive, transitive closure of $\to$. Notice that for a given language, there can be many different semantics. Most of the constructions that follow depend on our choices and would be (slightly) different had we chosen a different semantics.

We will define the *state space* as the graph whose vertices are the states and the edges are the transitions, between states. Here we briefly recall, and fix some notations for directed graphs that will be used throughout the book.

Definition 1.1.9. A *directed graph* $G = (V, E, \partial^-, \partial^+)$ consists of a set V of vertices (or states), a set E of edges and two functions $\partial^-, \partial^+ \colon E \to V$ respectively sending each edge to its source and target in V.

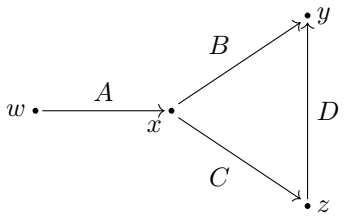
Definition 1.1.10. A *labelled graph* is a graph together with a set $\mathcal{L}$ of labels and a function $l \colon E \to \mathcal{L}$ associating a label to each edge.

We sometimes write $x \overset{A}{\to} y$ for the edge e_A such that $l(e_A) = A$, $\partial^-(e_A) = x$ and $\partial^+(e_A) = y$.

Example 1.1.11. We give a simple example of a graph G, where $V = \{w, x, y, z\}$ $E = \{e_A, e_B, e_C, e_D\}$ such that $l(e_\omega) = \omega$, and

$$w = \partial^-(e_A) \qquad x = \partial^+(e_A) = \partial^-(e_B) = \partial^-(e_C)$$
$$y = \partial^+(e_B) = \partial^+(e_D) \qquad z = \partial^+(e_C) = \partial^-(e_D)$$

We can represent such a graph in the classical way below, associating to each vertex a point and each edge an arrow from its source to it target



Definition 1.1.12. Given a graph G, a *path* π on G is

- a finite non-empty sequence $e_1 \cdot \cdots \cdot e_n$ of edges e_i of G such that $\partial^+(e_i) = \partial^-(e_{i+1})$,
- or a single vertex x corresponding to the empty path ε_x.

The source (resp. target) of π is the source of e_1 (resp. target of e_n). We denote the empty path on a vertex x as ε_x with source and target x.

Given two paths π, τ such that $\partial^-(\tau) = \partial^+(\pi)$, we write $\pi \cdot \tau$ for their concatenation.

Definition 1.1.13. Given a graph G, given two vertices x, y of G, we say that x is *reachable* from y when there exists a path with source x and target y.

Definition 1.1.14. Given a program P, the *state space* $\mathcal{G}_P$ of this program is the graph whose vertices are the states $\langle \sigma, c \rangle$ and edges are the reductions from Definition 1.1.8.

Definition 1.1.15. Given a program P, its *initial state* is the state $\langle \sigma_0, P \rangle$ with σ_0 defined in Definition 1.1.6. Its terminal states are any states of the form $\langle \sigma, \texttt{skip} \rangle$.

Definition 1.1.16. A *path* on $\mathcal{G}_P$ is a morphism of the free category $\mathcal{G}_P^*$ over the graph $\mathcal{G}_P$.

It is equivalent to a sequence $\pi = (\langle \sigma_i, c_i \rangle \to \langle \sigma_{i+1}, c_{i+1} \rangle)_{1 \leq i < n}$ of reductions. We write $\pi \colon \langle \sigma_1, c_1 \rangle \to^* \langle \sigma_n, c_n \rangle$ to say that π is a path from c_1 to c_n.

Definition 1.1.17. We call an *execution* of a command c a sequence of reductions $\langle \sigma_0, c \rangle \to^* \langle \sigma', c' \rangle$. We say that an execution is *maximal* for a command c when it is a path of the form $\langle \sigma_0, c \rangle \to^* \langle \sigma', \texttt{skip} \rangle$.

Definition 1.1.18. Given a program P, a state $\langle \sigma, c \rangle$ of $\mathcal{G}_P$ is said to be *reachable* when there exists an execution with $\langle \sigma, c \rangle$ as its target, i.e. there exists a path $\pi \colon \langle \sigma_0, P \rangle \to^* \langle \sigma, c \rangle$. Otherwise, it is said to be *unreachable.*

As each of the reductions in Definition 1.1.8, comes from the evaluation of at most single action or boolean expression, we can label each reduction by this evaluation. In all other cases (mostly when removing `skip` instructions), it is labelled by the empty word. The executions can then be represented by the sequence of actions and boolean conditions that are executed along the path. More precisely, they will be represented by words on the alphabet consisting of actions and boolean expression of the program, called the trace of an execution.

Definition 1.1.19. A *word* w of length $n > 1$ over an alphabet $\mathbb{A}$ is a finite sequence $w_1 \dots w_n$ of elements of $\mathbb{A}$. The empty word ε is the only word of length 0.

We write $\mathbb{A}^n$ for the set of words of length n and $\mathbb{A}^* = \bigcup_{n \in N} \mathbb{A}^n$ for the set of all words.

Definition 1.1.20. Given two words $w = w_1 \dots w_p$ and $w' = w'_1 \dots w'_q$ on an alphabet $\mathbb{A}^*$, their *concatenation* is

$$w \cdot w' = w_1 \dots w_p w'_1 \dots w'_q$$

We extend this to sets of words by setting

$$R \cdot R' = \{w \cdot w' \mid w, \in R, w' \in R'\} \approx R \times R'.$$

Where $R \times R'$ is the standard cartesian product of sets.

Definition 1.1.21. To each path π we associate a word $\mathrm{tr}(\pi)$ of $(\mathcal{X} \sqcup \mathcal{B})^*$, called the *trace* of π, where ε is the empty word

- When π corresponds to the one-step reduction $\dfrac{c \in \mathcal{X} \setminus \{\texttt{skip}\}}{\langle \sigma, c\rangle \to \langle [\![c]\!](\sigma), \texttt{skip}\rangle}$, $\mathrm{tr}(\pi) = c$.
- When π corresponds to a one-step reduction with $[\![b]\!](\sigma) = \top$ as premise, $\mathrm{tr}(\pi) = b$. Such a reduction corresponds to choosing a branch in a conditional branching or entering a `while` loop.
- When π corresponds to a one-step reduction with $[\![b]\!](\sigma) = \bot$ as premise, $\mathrm{tr}(\pi) = \neg b$. Such a reduction corresponds to exiting a `while` loop, or exploring the `else` branch of a conditional branching.
- For each reduction π, different from the ones above, deduced with a rule of Definition 1.1.8 with no premise we have $\mathrm{tr}(\pi) = \varepsilon$.
- For each reduction π deduced with a rule of Definition 1.1.8 with one reduction π' as premise, we have $\mathrm{tr}(\pi) = \mathrm{tr}(\pi')$.
- The word associated to a concatenation of reductions is the concatenation of their associated labels $\mathrm{tr}(\pi' \cdot \pi) = \mathrm{tr}(\pi') \cdot \mathrm{tr}(\pi)$.
- The empty word ε is associated to the empty path, i.e. $\mathrm{tr}(\varepsilon) = \varepsilon$.

When $\pi\colon \langle \sigma_0, c\rangle \to^* \langle \sigma, c'\rangle$ is an execution of c (Definition 1.1.17), we call $\mathrm{tr}(\pi)$ an *execution trace* of c. It is a *maximal* execution trace when π is a maximal execution. We often do not distinguish π and $\mathrm{tr}(\pi)$ and refer to both as execution traces.

We write $T(c, \sigma)$ the set of all traces of paths π starting from $\langle \sigma, c\rangle$, and simply $T(c)$ for the set of execution traces.

Example 1.1.22. Let us consider the program `(x:=0 || x:=1); if x==0 then` c_0 `else` c_1, where the == operator compares two integer values for equality. Because of parallelism, there are two family of maximal executions traces

$$\texttt{x:=1} \cdot \texttt{x:=0} \cdot \texttt{x==0} \cdot \mathrm{tr}(c_0)$$
$$\texttt{x:=1} \cdot \texttt{x:=0} \cdot \neg\texttt{x==0} \cdot \mathrm{tr}(c_1)$$

where $\mathrm{tr}(c_i)$ is a maximal execution trace of the command c_i.

Definition 1.1.23. Let $\pi\colon x \to y$ a path in $\mathcal{G}_P$. Let $l_\pi = (l_1, \dots, l_k)$ the subword of $\mathrm{tr}(\pi)$ obtained by removing all letters not included in $\mathcal{X}$. We define the *evaluation* of the path π as

$$[\![\pi]\!] = [\![l_k]\!] \circ \cdots \circ [\![l_1]\!](\sigma)$$

Definition 1.1.24. Two commands c_1, c_2 are *contextually equivalent*, written $c_1 \approx c_2$, when for every state $\sigma \in \Sigma$, we have $T(c_1, \sigma) = T(c_2, \sigma)$

Two contextually equivalent commands will have the same effect on the state and memory of the program, and can thus be simplified.

Proposition 1.1.25. *For all commands c, c', c'', the following commands are contextually equivalent:*

$$\texttt{skip};c \approx c \qquad (c\|c')\|c'' \approx c\|(c'\|c'')$$
$$\texttt{skip}\|c \approx c \approx c\|\texttt{skip} \qquad c\|c' \approx c'\|c$$

Proof. For the case $\texttt{skip}\|c \approx c$, it is proved by directly remarking that the reduction $\langle \sigma, \texttt{skip}\|c\rangle \to \langle \sigma, c\rangle$ does not have a premise, so the set of all traces will be the same.

All other contextual equivalences with a `skip` are treated similarly.

The case $(c\|c')\|c'' \approx c\|(c'\|c'')$ can be proved by remarking that the executions traces of $a\|b$ are the interweaving of the execution traces of a and b. □

Remark 1.1.26. In this language, the following classical equivalence holds:

$$\texttt{while } b \texttt{ do } c \approx \texttt{if } b \texttt{ then } (c\texttt{;while } b \texttt{ do } c)$$

Thus, when only considering verification up to a certain depth, loops can be unrolled into a finite number of conditional branching for the purposes of verification

1.1.3 Toward verification of programs

The most common properties we aim at verifying in programs can be regrouped in the two following categories:

1. *Functional properties* which describe compliance of the program with a mathematical specification. For instance, ensuring that an implementation of the factorial actually associates for each integer n as input, the factorial $n!$ as a result. These usually describe invariants of the program, expressed in proof-theoretic form [36] and generally verified using proof assistants, model-checking [8] or abstract interpretation [44].

2. *Reachability properties*, which correspond to checking that a set of forbidden positions/states X cannot be reached (Definition 1.1.18) from the initial state through a series of reductions (*execution of our program*). When there is no such execution trace from the initial state of our program to a state of X, we say that the program is *correct* w.r.t. the set X. The set of positions could be an error instruction in a program, and we want to guarantee that it will never be reached. Sometimes we will also be interested in knowing which states of a program can be reached.

In this thesis we will focus on the latter reachability properties which are a particular case of a liveness property (properties that stipulate that "something happens", detailed in [2]). A naive solution to verify the correctness of a program would be to explore all the reduction paths using our favourite tree exploration strategy (e.g. depth-first, breadth-first…). Then for every state reached during the exploration, we could check if it does belong to the set of forbidden positions. This, of course, is terribly inefficient, as we are essentially exploring all the possible schedulings of the program, and it may not even terminate when considering program with loops as shown in the Example 1.1.27.

Example 1.1.27. Let us consider the following program, computing the *Syracuse Sequence* starting from x, described by the command below.

```
x := n;
while x!=1 do
    if x mod 2 !=0 then
        x := 3*x +1
    else
        x := x/2
```

We want to check that this program terminates, i.e. the state $\langle x \mapsto 1, \texttt{skip} \rangle$ is always reached. Then we will need to check the infinite number of traces of the language on the following regular expression which describes all possible maximal executions:

$$(\texttt{x:=n}) \cdot \left((x! = 1) \cdot \big((\texttt{x//2!=0}) \cdot (\texttt{x:=3x+1}) \big) + \big(\neg(\texttt{x mod 2!=0}) \cdot (\texttt{x:=x/2}) \big) \right)^* \cdot \neg(\texttt{x!=1})$$

These problems come from the loops and are not specific to concurrent programs. For traditional sequential programs there already exist quite a number of techniques to deal with these problems (either by considering correctness properties of execution traces of bounded length and unrolling loops, or use widening operators associated to abstract interpretation domains [10]).

1.1.3.1 Verification of concurrent programs

In the rest of this thesis, we will focus on the verification of properties specific to concurrent programs, and more precisely, on reachability properties of such programs. In this context, one of the biggest problems we face is the *state space explosion problem* [7]: in order to prove that a program is correct, we have to check out all the possible execution traces coming from all the potential schedulings of the different threads. Even without loops, the number of traces generated by a parallel composition of threads is exponential in the size of the program as shown in the Example 1.1.28.

Example 1.1.28. Consider the following program $P = A \| \dots \| A$, composed of n copies of threads that are a single action A. Finding a scheduling amounts to finding an order on these n copies. Thus, there are $n!$ different schedulings, and as many execution traces.

Definition 1.1.29. A state $\langle \sigma, c \rangle$ of a program P is called:

- *unreachable*, if there are no execution traces from the initial state to this state,

- *deadlocked* when the state $\langle \sigma, c \rangle$ does not correspond to a terminal state of the program (Definition 1.1.15) and there is an execution from the initial state to the state $\langle \sigma, c \rangle$ that is not a proper prefix of another execution trace,
- *unsafe* when there is an execution trace with $\langle \sigma, c \rangle$ as target which is the prefix of an execution trace with a deadlock as target,
- *doomed* when there is an execution trace with $\langle \sigma, c \rangle$ as target which is not a prefix of an execution trace reaching the terminal state of the program.

Unreachable positions are positions that may never be reached during the execution of a program. These don't seem to be problematic at first, if they are never reached, they should not pose any problem. But these positions are witnessing the presence of dead code, that will never be executed. In critical systems, every line of code is there for a reason, so there should be no such unreachable positions. Finding these positions can help detect a misconception from the part of the programmer.

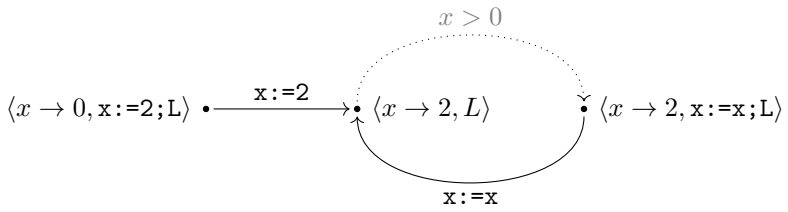
Deadlocks pose a much bigger problem. They are positions of the program in which it is blocked, or "frozen", and cannot progress at all. This situation generally arises when two or more threads are waiting on each other to free up a resource before they do anything else. They are locked in place and functionally dead, hence the term "deadlock" [13].

Finally, *unsafe* positions are the positions from which it is possible to eventually reach a deadlock and *doomed* positions, are the positions that can only reach deadlocks or loop on forever. This precision is important as we will eventually be considering programs with loops.

Definition 1.1.30. A program P is *safe* when its initial state is not unsafe.

In such a program P, there are no execution traces that reach a deadlock. There might be other problems as shown in the Example 1.1.31 below. The term *safe* here is specifically for concurrent related problems and might miss some problems that are specific to sequential programs. It is good to keep this in mind going forward.

Example 1.1.31. Let's look at the classical blunder of programming: non-terminating loops. The program $P =$`x:=2 ; while x>0 do x:=x` is technically safe. Even though it will be stuck in the `while` loop. If we look at the set of states and transitions, writing $L =$`while x>0 do x:=x` for the loop:



We see that all executions traces are words of the language $\texttt{x:=2} \cdot (x > 0 \cdot \texttt{x:=2})^*$ and thus can always be extended. The program is thus safe in the sense of Definition 1.1.30 even though a program that loops indefinitely might not be the desired result.

Remark 1.1.32. Deadlocks, unsafe and doomed positions are specific to concurrent programs. Sequential programs can be shown to not have deadlock positions. An example of such positions is given later on in Example 1.2.37.

Remark 1.1.33. For deadlocks (and similarly for unsafe and doomed regions) we do not require all executions reaching our deadlock to not have an extension. A point where any execution cannot be extended is considered a deadlock. This choice of definition is because we will, in the following consider programs where the properties of a position does not depend on the execution trace leading to it (Definition 1.2.23).

Remark 1.1.34. In the rest of this section, we will consider only finite executions. There are a lot of subtle differences that can arise from the presence of infinite executions (such as executions that are stuck in a loop, which would not count as deadlocks). More generally loops bring a lot of problems to the table that we will try to deal with.

The search for deadlocks, unsafe and doomed positions will be the main goal of the following chapters, as well as dealing with the state explosion problem.

1.2 Control flow graphs

In this section, we present a first model of programs based on a graph representation of the structure of our program: the transition graph (or a control-flow graph) [1]. This classical construction allows us to abstract away from the syntax of the programming language, and present in a simple fashion the subtleties of concurrent languages.

1.2.1 Transition graphs

The first model we introduce is based on directed graphs. Their definition was previously introduced in Definition 1.1.9. In the following, we will need some additional operations in order to combine graphs, to represent more elaborate programs.

Definition 1.2.1. Given three graphs $G_1 = (V, E, \partial^-, \partial^+)$, $G_1 = (V_1, E_1, \partial_1^-, \partial_1^+)$ and $G_2 = (V_2, E_2, \partial_2^-, \partial_2^+)$. We define the following constructions:

- The disjoint union $G = G_1 \sqcup G_2 = (V_1 \sqcup V_2, E_1 \sqcup E_2, \partial^-, \partial^+)$ where

$$\partial^-(e) = \begin{cases} \partial_1^-(e) \text{ if } e \in E_1 \\ \partial_2^-(e) \text{ if } e \in E_2 \end{cases} \qquad \partial^+(e) = \begin{cases} \partial_1^+(e) \text{ if } e \in E_1 \\ \partial_2^+(e) \text{ if } e \in E_2 \end{cases}$$

- The tensor product $G_1 \otimes G_2 = (V_1 \times V_2, E_1 \times V_2 \sqcup V_1 \times E_2, \partial^-, \partial^+)$ where

$$\partial^-|_{E_1 \times V_2} = \partial_1^- \times \mathrm{id} \qquad \partial^+|_{E_1 \times V_2} = \partial_1^+ \times \mathrm{id}$$
$$\partial^-|_{V_1 \times E_2} = \mathrm{id} \times \partial_2^- \qquad \partial^+|_{V_1 \times E_2} = \mathrm{id} \times \partial_2^+$$

- Given $x, y \in V$, the quotient graph $G[x = y]$ is the graph obtained by identifying the vertices x and y in G
- Given $V' \subseteq V$, the induced subgraph $G|_{V'}$ is the restriction of the graph G to the vertices of V' i.e.

$$G|_{V'} = (V', E' = \{e \in E \mid \partial^-(e) \in V' \text{ and } \partial^+(e) \in V'\}, \partial^-|_{E'}, \partial^+|_{E'})$$

As we previously stated, each of the commands in $\mathcal{C}$ give rise to a structure in the code and allows us to easily formalize the notion of a transition graph (or control-flow graph) associated to a program of our language.

Definition 1.2.2. The *transition graph* $G_c = (G_c, l_c, s_c, t_c)$ associated to a command c is a graph G_c with labels in the set $\mathcal{L} = \mathcal{X} \coprod \mathcal{B}$, equipped with two distinguished vertices $s_c, t_c \in E$ called the *start* and *end*. It is defined inductively as follows.

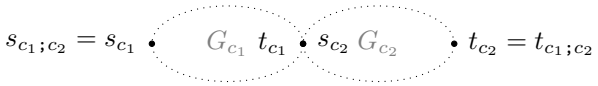
- $G_{\texttt{skip}}$ is the graph with a single vertex and no edge:

$$s_{\texttt{skip}} \bullet t_{\texttt{skip}}$$

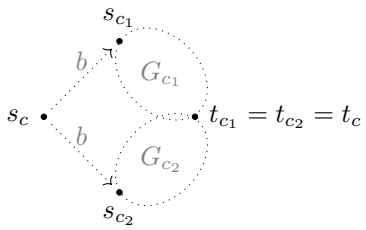
- G_A with $A \in \mathcal{X}$ is the graph with two vertices and a single edge from s_A to t_A labelled A.

$$s_A \bullet \xrightarrow{A} \bullet\, t_A$$

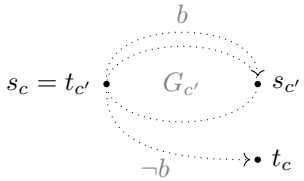
- $G_{c_1;c_2} = (G_{c_1} \sqcup G_{c_2})[t_{c_1} = s_{c_2}]$ is the graph obtained by identifying t_{c_1} and s_{c_2}.



- G_c, with $c = \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ is the following graph, obtained by disjoint union of G_{c_1} and G_{c_2} adding a new vertex s_c and two labelled transitions b (resp. $\neg b$) from s_c to s_{c_1} (resp. s_{c_2}). Finally we identify t_{c_1} and t_{c_2}.



- G_c, with $c = \texttt{while}\ b\ \texttt{do}\ c'$ is obtained from $G_{c'}$ by adding a vertex t_c, adding an edge from $t_{c'}$ to t_c albelled $\neg b$, and adding an edge $t_{c'}$ to $s_{c'}$ labelled b, and $s_c = t_{c'}$.



- $G_{c_1 \parallel c_2} = G_{c_1} \otimes G_{c_2}$. where $s_{c_1 \parallel c_2} = (s_{c_1}, s_{c_2})$ and $t_{c_1 \parallel c_2} = (t_{c_1}, t_{c_2})$

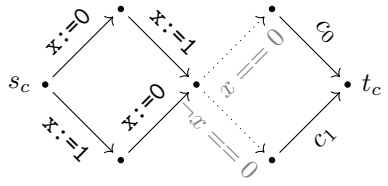
We call a vertex of G_P a position of the program P.

The edges labelled by conditions in transition graphs are drawn with dotted arrows to distinguish them from those labelled by actions. This is only a drawing convention; there is no difference between the two types of edges except the sets in which they are labelled.

Example 1.2.3. Let us revisit our program of Example 1.1.22:

$$\texttt{(x:=0 || x:=1); if x==0 then } c_0 \texttt{ else } c_1$$

For now let us suppose that c_0 and c_1 are actions in $\mathcal{X}$. It has the following transition graph:



If we count the paths, there are four different paths from s_c to t_c, whereas there are only two maximal execution traces.

Remark 1.2.4. For a program P, by construction, there is always a path from s_P to any of its positions p and from p to t_P.

There are usually more paths than execution traces as defined in Definition 1.1.21, as shown in the Example 1.2.3 above. Nonetheless, there is a link between execution traces and paths on the graph starting at s_P, but some refinement must be made.

Definition 1.2.5. A *potential execution* of a program P is a path in G_P starting from s_P. It is maximal when its target is t_P

Definition 1.2.6. Given a potential execution π of a program P, the associated *potential execution trace* $\mathrm{tr}(\pi)$ is defined as the concatenation of the labels of π.

Determining which traces are actual executions traces of our program and will occur during its execution depends on the semantics we chose for our programming language, as formalized in Definition 1.1.6. In our case, it comes down to verifying that all boolean conditions are verified.

Definition 1.2.7. Given a potential execution π of a program P we say that it is *valid* when its trace $\mathrm{tr}(\pi)$ is of the following form:

- the empty word ε.
- Or $\mathrm{tr}(\pi') \cdot A$, with $A \in \mathcal{X}$ and π' a valid execution.
- Or $\mathrm{tr}(\pi') \cdot b$, with $b \in \mathcal{B}$, π' a valid execution and $[\![b]\!] \circ [\![\pi']\!](\sigma_0) = \top$.

Definition 1.2.8. Given a program P define the *execution traces* $T(P)$ of our program P as the set of all valid potential traces.

Example 1.2.9. Consider the following program

$$\texttt{while } b \texttt{ do } A\texttt{; if } b \texttt{ then } B \texttt{ else } C$$

Its set of potential executions are the paths with labels in the language associated to the regular expression:

$$\big((b \cdot A)^* \cdot \neg b \cdot b \cdot B\big) + \big((b \cdot A)^* \cdot \neg b \cdot \neg b \cdot C\big)$$

The boolean conditions appearing in those traces should be thought of assumptions on the state of the memory, under which the path is valid. This is why they are called potential. Here it is easy to determine which are the maximal executions traces. As $\neg b$ and b cannot both hold without any effect on the memory in between, all traces in the sub-language $(b \cdot A)^* \cdot \neg b \cdot b \cdot B$ cannot be maximal.

1.2.2 Introducing resources

As we have seen in Example 1.1.22, the execution of a concurrent program can be non-deterministic, which is a problem unique to these types of programs. This problem is not only linked to the scheduling of the actions as previously seen. Let us consider the following program in which two threads try to access and increment the same shared resource:

$$\texttt{x:=0;}\,(\texttt{x:=x+1} \parallel \texttt{x:=x+1}) \tag{1.1}$$

In a weak memory model (when sequential consistency is not enforced), this is an unspecified outcome [51]. Indeed, one might first think that at the end of the execution x will be equal to 2, but because of how memory access is implemented in practice, the variable could contain 1 or a completely unrelated value.

Example 1.2.10. If we suppose that `x:=x+1` is not atomic, for example if fetching x from memory and writing to x can be interleaved with other actions we could have the following "trace" for the 1.1:

1. The first thread fetches the value of x ($= 0$) and writes it to a new fresh temporary variable
2. The second thread fetches the value of x ($= 0$) and writes it to a new fresh temporary variable
3. First and second thread increment there respective fresh variable (both equal to 1)
4. First thread writes 1 to `x`
5. Second thread writes 1 to `x`

Which would give $x = 1$ at the end of the execution.

In order to avoid such unpredictable behaviour when using shared memory, most systems come with constructions called *mutex* restricting access to portion of codes. Given a mutex a, a thread may perform the following operations [11], called synchronization primitives:

- lock the associated *resource*, modelled by the instruction $\texttt{P}_a$
- unlock the associated *resource*, modelled by the instruction $\texttt{V}_a$

The system then guarantees that the mutex may not be locked more than once. Threads attempting to lock an already locked thread will be "frozen" and will not progress further until the resource is unlocked, and they manage to lock it themselves. Thus, to guarantee the predictability of the behaviour of the program of 1.1, it should be re-written:

$$\texttt{x:=0};\big((\texttt{P}_a;\texttt{x:=x+1};\texttt{V}_a)\,\|\,(\texttt{P}_a;\texttt{x:=x+1};\texttt{V}_a)\big)$$

Mutexes ensure that the sequence of instruction between $\texttt{P}_a$ and $\texttt{V}_a$, called a blocking section, is atomic, i.e. it will not be interleaved with instruction from another thread using the mutex a in the same way. For other examples of programs with resources we refer to [15].

Remark 1.2.11. The uses of mutexes do not decrease the number of execution traces of the Example 1.1.22. This non-determinism is intrinsic to parallel programs.

We extend our language (Definition 1.1.1) with a set $\mathcal{R} = \{a, b, \dots\}$ of resources (or semaphores), together with a function $\kappa\colon \mathcal{R} \to \mathbb{N}$ associating each resource a to its *capacity* (the number of times it can be locked) $\kappa_a \in \mathbb{N}$.

Definition 1.2.12. We extend the syntax of **PIMP** programs with two additional actions in $\mathcal{X}$ for each resource in $\mathcal{R}$:

$$c_{act} \quad ::= \quad \mathcal{X} \quad | \quad \texttt{P}_a \quad | \quad \texttt{V}_a$$

Remark 1.2.13. This also extends the sets of labels and constructions rules for the transition graph G_P in accordance with the rules defined for actions:

$$s_{\texttt{P}_a} \bullet\xrightarrow{\texttt{P}_a}\bullet\ t_{\texttt{P}_a} \qquad s_{\texttt{V}_a} \bullet\xrightarrow{\texttt{V}_a}\bullet\ t_{\texttt{V}_a}$$

Remark 1.2.14. Mutexes are a particular case of semaphores where $\kappa_a = 1$.

We must also extend the operational semantics of our language introduced in Definition 1.1.6 to encompass this new notion.

Definition 1.2.15. A *state* σ for the language **PIMP** is now the data of two functions $\sigma_v\colon \mathsf{Var} \to \mathbb{Z}$ and $\sigma_r\colon \mathcal{R} \to \mathbb{Z}$. Where σ_v (resp. σ_r) associates to each variable (resp. resource) its content (resp. availability). The initial state σ_0 is the function associating to each variable and each resource 0. We write $\Sigma = \mathbb{Z}^{\mathsf{Var}} \times \mathbb{Z}^{\mathcal{R}}$ for the set of states.

Remark 1.2.16. The assumption that resources are at their maximum capacity in the initial state can be made without loss of generality. Indeed, if for $a \in \mathcal{R}$, $\sigma_0(a) > 0$, studying P is the same as studying the program $P' = \underbrace{\texttt{P}_a;\dots;\texttt{P}_a}_{\sigma_0(a)\text{ times}};P$ where the program P' starts at maximum capacity for a.

Given $x \in \mathsf{Var}$ (resp. $a \in \mathcal{R}$), we will often write $\sigma(x)$ (resp. $\sigma(a)$) for $\sigma_v(x)$ (resp. $\sigma_r(a)$) when the context makes it clear.

Definition 1.2.17. Let $x \in \mathsf{Var}, e \in \mathcal{A}, a \in \mathcal{R}$. Given a state $\sigma \in \Sigma$. The previous Definition 1.1.7 of the evaluation for the following case is replaced by:

$$[\![\texttt{x := e}]\!]_{\mathcal{X}}(\sigma) = (\sigma_v[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)], \sigma_r)$$

The evaluation of the actions $\mathtt{P}_a$ and $\mathtt{V}_a$ are defined as follows:

$$[\![\mathtt{P}_a]\!]_{\mathcal{X}}(\sigma_v, \sigma_r) = (\sigma_v, \delta_a^+(\sigma_r)) \qquad [\![\mathtt{V}_a]\!]_{\mathcal{X}}(\sigma_v, \sigma_r) = (\sigma_v, \delta_a^-(\sigma_r))$$

where $\delta_a^+(\sigma)(a) = \sigma(a)+1$, $\delta_a^-(\sigma)(a) = \sigma(a)-1$, and $\delta_a^+(\sigma)(b) = \delta_a^-(\sigma)(b) = \sigma(b)$ if $b \neq a$.

Definition 1.2.18. Let $a \in \mathcal{R}$. We extend the reduction relation with special rules for the actions $\mathtt{P}_a$ and $\mathtt{V}_a$ which overrides the previous rules in Definition 1.1.8

$$\frac{\sigma_r(a) < \kappa_a}{\langle \sigma, \mathtt{P}_a \rangle \to \langle [\![\mathtt{P}_a]\!](\sigma), \mathtt{skip} \rangle} \qquad \frac{\sigma_r(a) > 0}{\langle \sigma, \mathtt{V}_a \rangle \to \langle [\![\mathtt{V}_a]\!](\sigma), \mathtt{skip} \rangle}$$

The notion of validity of Definition 1.2.7 is extended accordingly to express that in a valid execution trace, the locked resources have to be available and one cannot add more instances of a resource of a resource than its capacity.

Definition 1.2.19. Given a potential execution π of a program P with resources, we say that it is *valid* when its trace $\mathrm{tr}(\pi)$ is of the following form:

- the empty word ε.
- Or $\mathrm{tr}(\pi') \cdot \mathtt{P}_a$, with $a \in \mathcal{R}$, π' a valid execution and $[\![\pi']\!](\sigma_0)(a) < \kappa_a$.
- Or $\mathrm{tr}(\pi') \cdot \mathtt{V}_a$, with $a \in \mathcal{R}$, π' a valid execution and $[\![\pi']\!](\sigma_0)(a) > 0$.
- Or $\mathrm{tr}(\pi') \cdot A$, with $A \in \mathcal{X} \setminus \{\mathtt{P}_a, \mathtt{V}_a\}$ and π' a valid execution.
- Or $\mathrm{tr}(\pi') \cdot b$, with $b \in \mathcal{B}$, π' a valid execution and $[\![b]\!] \circ [\![\pi']\!](\sigma_0) = \top$.

Remark 1.2.20. The primitives $\mathtt{P}_a$ and $\mathtt{V}_a$ could have been implemented as features of the language, but it is much more complicated [11]. Similarly, we could have restricted to mutex without loss of generality.

Remark 1.2.21. We have chosen semaphores as our synchronization primitives. There are many other such mechanisms used in practice (e.g. monitors, barriers,…) used to simplify implementation of some structures. As they can be modelled by semaphores, we do not lose any generality in our approach.

1.2.3 Conservative programs

Consider the program `if` b `then` $\mathtt{P}_a$ `else skip`. Depending on the maximal path we follow, the number of times the resource a was locked/released depends on whether b is true or not. As we do not want to have to consider the whole trace space when verifying the program we will restrict our study to programs where the consumption of resources does not depend on the path, but only on its endpoints.

Definition 1.2.22. The *consumption* of a path π w.r.t. a resource $a \in \mathcal{R}$ and a state σ is defined as $[\![\pi]\!](\sigma)(a) \in \mathbb{Z}$.

Definition 1.2.23. A program P is *conservative* when for any pair of paths $\pi, \tau \colon x \to y$ in G_P with same source and target we have for any state $\sigma \in \Sigma$ and any resource $a \in \mathcal{R}$:

$$[\![\pi]\!](\sigma)(a) = [\![\tau]\!](\sigma)(a)$$

Definition 1.2.24. Given a conservative program P, its initial state σ_0 and a vertex x in G_P, we define the *consumption* of x, written $[\![x]\!] \colon \mathcal{R} \to \mathbb{Z}$ as the restriction of $[\![-]\!]$ to the set of resources $\mathcal{R}$:

$$[\![x]\!] = [\![\pi]\!](\sigma_0)|_{\mathcal{R}}$$

for some path $\pi \colon s_P \to x$

The above definition is well-defined. Indeed, the chosen path π does not matter as the program P is explicitly chosen to be conservative and all vertexes of G_P are reachable by Remark 1.2.4. Furthermore, for any resource a, $[\![x]\!](a)$ corresponds to its consumption w.r.t. the resource a from Definition 1.2.22.

Definition 1.2.25. Given a conservative program P, a vertex of its transition graph G_P is *valid* (and otherwise *forbidden*) when for every $a \in \mathcal{R}$, we have $O \leq [\![x]\!](a) \leq \kappa_a$.

Proposition 1.2.26. *Given a conservative program P and a path $\pi \colon x \to^* y$ in G_P, a state $\sigma \in \Sigma$ and a resource $a \in \mathcal{R}$, then $[\![\pi]\!](\sigma)(a) = \sigma(a) + [\![y]\!](a) - [\![x]\!](a)$.*

Proof. See [18, Proposition 3.12]. □

This shows that, in a conservative program, the number of times a resource has been locked during a potential execution path π depends only on the target of π.

Proposition 1.2.27. *Given a conservative program P and an execution trace $\pi \colon s_P \to x$ on G_P. Then x is a valid vertex, in the sense of Definition 1.2.25.*

Proof. See [18, Proposition 3.15]. □

For conservative programs, we can define a notion of global consumption, directly on the syntax.

Definition 1.2.28. The *consumption* of a program P is the partial function $\Delta(P) : \mathcal{R} \to \mathbb{Z}$ defined by induction on P by

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

$$\Delta(P;Q) = \Delta(P \parallel Q) = \Delta(P) + \Delta(Q)$$
$$\Delta(\texttt{if } b \texttt{ then } P \texttt{ else } Q) = \Delta(P) \quad \text{if } \Delta(P) = \Delta(Q)$$
$$\Delta(\texttt{while } b \texttt{ do } P) = \underline{0} \quad \text{if } \Delta(P) = \underline{0}$$

where $\underline{0}$ is the constant function equal to 0 and δ_a the indicator function of a.

Proposition 1.2.29. *The function Δ is only partially defined on programs. A program P is conservative if and only if $\Delta(P)$ is well-defined.*

Proof. See [18, Proposition 3.8]. □

The resource consumption $\Delta(P)$ gives the amount of times each resource has been locked or released, i.e. the difference between the global number of instructions $\mathtt{P}_a$ and $\mathtt{V}_a$ encountered in an execution of P.

Example 1.2.30. Given a single mutex a, the program `if` b `then` $\mathtt{P}_a$ `else skip` is not conservative. The two paths $b{\cdot}\mathtt{P}_a$ and $\neg b{\cdot}\mathtt{skip}$ from s_P to t_P have a different consumption. Indeed,

$$[\![b \cdot \mathtt{P}_a]\!](a) = 1 \qquad\qquad [\![\neg b \cdot \mathtt{skip}]\!](a) = 0$$

Remark 1.2.31. Checking that a program is conservative is thus linear in the size of the program as it suffices to check $\Delta(P)$.

1.2.4 Pruned transition graph

In the rest of the thesis, we will always suppose that the program we are considering are conservative. By Proposition 1.2.27, no valid execution trace visits an invalid state. Thus, we can remove all invalid states from the transition graph, and any associated vertices without removing any valid paths. This gives us the *pruned transition graph* defined below.

Definition 1.2.32. The *pruned transition graph* $\overline{\mathcal{G}}_P$ is obtained from the transition graph G_P by removing all invalid vertices (in the sense of Definition 1.2.25), keeping the terminal position t_P. Formally, with V'_P the set of valid vertices,

$$\overline{\mathcal{G}}_P = \begin{cases} G_P|_{V'_P} & \text{if } t_p \in V'_P \\ G_P|_{V'_P} \sqcup t_P & \text{if } t_p \notin V'_P \end{cases}$$

With t_P the graph with a single vertex and no edge.

Remark 1.2.33. For future definitions, a program needs an initial and terminal position. That is why we ensure that t_P is not removed in Definition 1.2.32. As the starting position is always valid, such concerns are unnecessary for the s_P.

Lemma 1.2.34. *The inclusion $\overline{\mathcal{G}}_P \hookrightarrow G_P$ induces a bijection between valid paths from the initial vertex (i.e. execution traces) in both graphs.*

Proof. See [18, Lemma 3.18]. □

The Remark 1.2.4 about reachability of all vertices no longer holds in the pruned transition graph. This allows us to discover the troublesome positions described in Definition 1.1.29.

Proposition 1.2.35. *Given a conservative program P:*

- *Let $x \in \overline{\mathcal{G}}_P$ such that there is no path $s_P \to^* x$. Then x is unreachable.*
- *Let $x \in \overline{\mathcal{G}}_P \setminus \{t_P\}$. If x is reachable and there is no edge in $\overline{\mathcal{G}}_P$ with x as its source, then x is a deadlock*
- *Let $x \in \overline{\mathcal{G}}_P$, such that there is a path $s_P \to^* x \to^* y$ where y is a deadlock, then x is unsafe.*
- *Let $x \in \overline{\mathcal{G}}_P$, such that there is a path $s_P \to^* x$ and no path $x \to^* t_P$. Then x is doomed.*

Proof. See [18, Proposition 3.19]. □

The above Proposition 1.2.35 allows us to define a less naive verification method for the properties described in Definition 1.1.29.

Proposition 1.2.36. *Properties of Definition 1.1.29 can be discovered from the pruned transition graph $\overline{\mathcal{G}}_P$ of the program P as follows:*

- *a position which is not reachable in $\overline{\mathcal{G}}_P$ from the beginning position is unreachable*
- *a position in $\overline{\mathcal{G}}_P \backslash \{t_P\}$ which is not unreachable and from which there is no transition is a deadlock*
- *a position from which there is a path to a potential deadlock is unsafe*
- *a position from which there is no path to t_p is doomed*

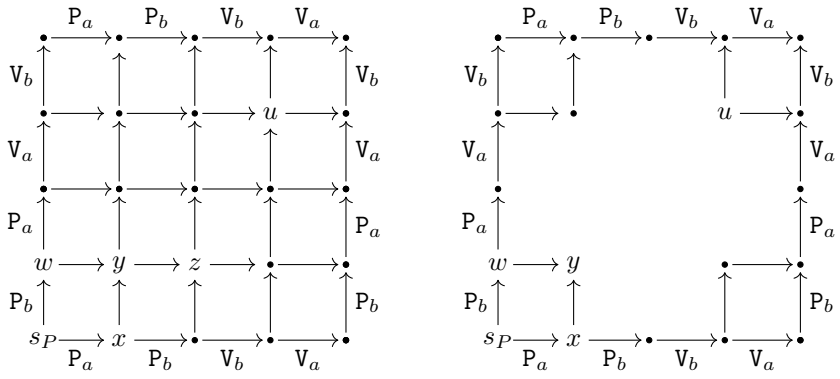
Proof. See [18, Proposition 3.19]. □

In the Proposition 1.2.36 we have to consider the position as "potential" candidates as they might be unreachable in the sense of Definition 1.1.29 (even though they might be reachable in $\overline{\mathcal{G}}_P$). Unreachable positions in $\overline{\mathcal{G}}_P$ are a subset of all unreachable positions, as shown in Remark 1.2.38, because of properties other than the structure of the program (i.e. how semaphores are used). This algorithm is still very inefficient, as it searches through all the vertices of the graph. Much of our focus in the thesis will be on methods of representing these valid positions in a more compact way.

Example 1.2.37. Let us consider the program P with two mutexes a, b of capacity $\kappa_a = \kappa_b = 1$:

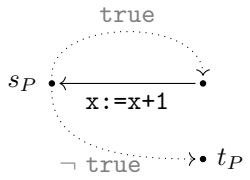
$$(\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$$

On the left we see its transition graph, and on the right its pruned transition graph (only exterior edges are labelled). The transition $s_P \to x$ is labelled by $\mathtt{P}_a$ which has the effect of increasing the consumption of a. Thus, $[\![x]\!](a) = 1 = \kappa_a$. Similarly, the transition $x \to y$ is labelled by $\mathtt{P}_b$ which increments the consumption of b. Then $[\![y]\!](b) = 1 = \kappa_b = \kappa_a = [\![y]\!](a)$. Then, by the same argument on the transition $y \to z$ we can see that $[\![z]\!](a) = 2 > \kappa_a$, which implies that z is not valid. A similar argument may be used for the rest of the pruned vertices.



Then y is a deadlock as it is not t_P and there are no transitions from y. The points x, w, s_P are unsafe as there is a path to y from these vertices. The vertex u is unreachable, and no positions, except for the deadlock, are doomed.

Remark 1.2.38. Not all undesirable positions are discovered by the Proposition 1.2.36. Indeed, let us consider the program $P =$ `while true do x:=x+1`. Its pruned transition graph can be represented as:



The position t_p is unreachable because the condition $\neg$`true` is never true. We could prune more by removing vertices following branching that are never satisfied, but it would make pruning our graph undecidable [18]. Furthermore, the non-reachability of these point does not come from concurrent program features, so we will not dwell on it more than necessary.

1.3 Directed geometric semantics

1.3.1 Asynchronous semantics

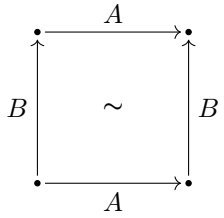
As explained in Section 1.1.3, the number of paths one has to check in order to verify a program might be exponential in the size of the program. To lower this number, we take an approach, historically called *true concurrency*, of considering schedulings of all actions up to a certain notion of commutation of these actions. Two actions commute when their effect on any state does not depend on the order in which they are executed.

Definition 1.3.1. We write $\mathfrak{S}_n$ the *symmetric group* of order n composed of all the permutations of the elements of the set $\{1, \dots, n\}$.

Definition 1.3.2. Given n actions $A_1, \dots, A_n$, we say that $A_1, \dots, A_n$ *commute* if for any permutation $\sigma \in \mathfrak{S}_n$ we have:

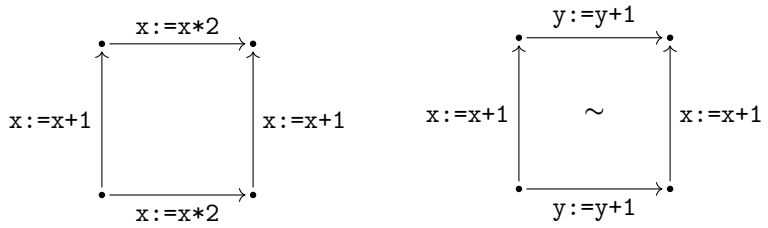
$$[\![A_1]\!] \circ \cdots \circ [\![A_n]\!] = [\![A_{\sigma(1)}]\!] \circ \cdots \circ [\![A_{\sigma(n)}]\!]$$

When two actions commute, we indicate this on their transition graph in the following way, by writing $\sim$ in the centre of the square formed by these paths:



Example 1.3.3. The two actions `x:=x+1`, `x:=x*2` do not commute. Indeed, $[\![\texttt{x:=x+1}]\!] \circ [\![\texttt{x:=x*2}]\!](\sigma_0)(x) = 1$ which is different from $[\![\texttt{x:=x*2}]\!] \circ [\![\texttt{x:=x+1}]\!](\sigma_0)(x) = 2$.

On the contrary, it is easy to check that `y:=y+1` and `x:=x+1` commute.



Remark 1.3.4. The only paths we are interested in verifying are the execution traces. Thus, the "correct" notion of commutation we would like to study would be a looser one: Two actions A, B commute when for each path $\pi : s_P \to x$, we have:

$$[\![B]\!] \circ [\![A]\!] \circ [\![\pi]\!](\sigma_0) = [\![A]\!] \circ [\![B]\!] \circ [\![\pi]\!](\sigma_0)$$

Unfortunately, this notion of commutativity is undecidable. Moreover, commutation in the sense of Definition 1.3.2 implies this commutation, which makes it a suitable over-approximation.

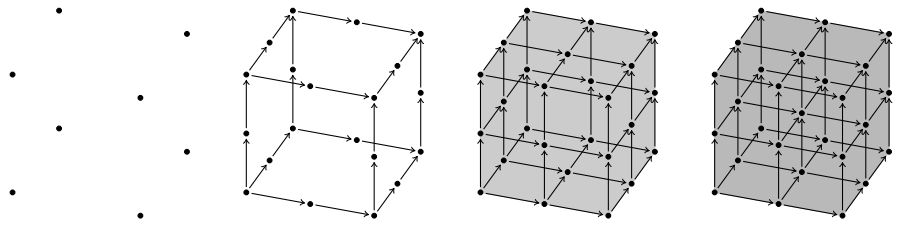
In the following Section 1.3.1.1, we will enhance the transition graph model by filling in the cubes of the graph to take into account this notion of commutation as in the Example 1.3.5 below. To do this we introduce precubical sets and new semantics.

Example 1.3.5. Consider the following program

$$P = \mathtt{P}_a; \mathtt{V}_a \| \mathtt{P}_a; \mathtt{V}_a \| \mathtt{P}_a; \mathtt{V}_a$$

Depending on the capacity κ_a of the resource a,

- If $\kappa_a = 0$ then, there are no transitions between states, and thus the semantics consists of 8 disjoint vertices.
- If $\kappa_a = 1$, then the transition graph is the skeletal cube, and there are no commutation/independence tiles.
- If $\kappa_a = 2$, then each smaller subdivided face has a commutation tile.
- If $\kappa_a = 3$ then we should fill the whole cube with commutation tiles.



$\kappa_a = 0$ $\kappa_a = 1$ $\kappa_a = 2$ $\kappa_a = 3$

1.3.1.1 Cubical semantics

A precubical set consists of sets of n-dimensional cubes, for all $n \in \mathbb{N}$, together with the data of their faces (i.e. which $n-1$ dimensional cube of the precubical set correspond to one of its face): each cube has two faces in each direction i with $0 \leq i < n$, a front face and a back face. We would also like these higher order cubes to encode notions of commutations presented in the start of this section.

Definition 1.3.6. A *precubical set* C consists of a family $(C_n)_{n\in\mathbb{N}}$ of sets whose elements are called n-cubes together with maps

$$\partial^+_{n,i} \colon C_n \to C_{n-1} \text{ and } \partial^-_{n,i} \colon C_n \to C_{n-1}$$

for all indices $n, i \in \mathbb{N}$ such that $0 \leq i < n$. $\partial^+_{n,i}$ and $\partial^-_{n,i}$ respectively associate each n-cube to its back and front face in the i-th direction such that for $0 \leq i \leq j < n$ and $\alpha, \beta \in \{+, -\}$

$$\partial^\beta_{n,j} \circ \partial^\alpha_{n+1,i} = \partial^\alpha_{n,i} \circ \partial^\beta_{n+1,j} \tag{1.2}$$

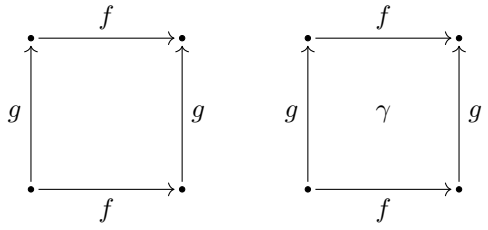
The 0-cubes and 1-cubes of a precubical sets are often called its *vertices* and its *edges* respectively. Given a set $\mathcal{L}$ of labels, a *labelled precubical set* (C, l) consists of a precubical set C equipped with a function $l \colon C_1 \to \mathcal{L}$ associating to each 1-cube (edges) a label and such that

$$l \circ \partial^-_{2,0} = l \circ \partial^+_{2,0} \quad \text{and} \quad l \circ \partial^-_{2,1} = l \circ \partial^+_{2,1}$$

Example 1.3.7. A labelled precubical set C, where $C_i = \emptyset$ for all $i \geq 2$, is equivalent to a labelled graph:

- C_0 can be seen as the set of its vertices,
- C_1 can be seen as the set of its edges,
- $\partial^+_{1,0}, \partial^-_{1,0}$ are respectively the source and target functions.

If we have 2-cubes, i.e. $C_2 \neq \emptyset$, then its elements are cells differentiating the empty square on the left (no 2-cubes) to the "filled" square on the right (corresponding to a 2-cube γ):



All 2-cubes of our precubical sets are of this form by the conditions on the labelling function l. We will use n-cubes to indicate that the actions labelling the 1-cubes which are used in their construction commute.

There is a nice categorical structure that will be of technical interest later on.

Definition 1.3.8. If we define the *cubical category* $\square$ to be the opposite of the category whose objects are natural numbers and whose morphisms are the face maps $\partial^{\alpha}_{n,i} \colon n \to n-1$. Then the category of precubical sets $\widehat{\square}$ can be defined as the category of presheaves over $\square$, i.e. the category of functors $\square^{op} \to \mathbf{Set}$.

Remark 1.3.9. There is a full and faithful embedding of the category of transition graphs into the category of precubical sets [53] (cf Example 1.3.7).

Definition 1.3.10. Given two cubical sets C and D, the operations previously defined on graphs can be extended to precubical sets, while coinciding with previous operations on the embedding of transition graphs.

- The disjoint union $C \sqcup D$ is the precubical set defined by

  $$(C \sqcup D)_n = C_n \sqcup D_n$$

  with face maps induced by those of C and D

- The tensor product $C \otimes D$ is the precubical set defined by

  $$(C \otimes D)_n = \coprod_{i+j=n} C_i \times D_j$$

  with face maps $\partial^{\alpha}_{n,k} \colon (C \otimes D)_n \to (C \otimes D)_{n-1}$ defined on $(x, y) \in C_i \times D_j$ such that $i + j = n$ by:

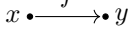
  $$\partial^{\alpha}_{n,k}(x,y) = \begin{cases} (\partial^{\alpha}_{n,k}(x), y) & \text{if } 0 \leq k < i \\ (x, \partial^{\alpha}_{n,k-i}(y)) & \text{if } i \leq k < n \end{cases}$$

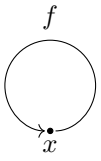
- The quotient $C[c = c']$ is the precubical sets obtained by identifying the 1-cubes c and c'.

- The restriction $C|_{C'_0}$ of a precubical sets to a subset $C'_0 \subseteq C_0$ is defined as the precubical sets where we remove from each C_n all cubes c_n such that there exists $\partial^{\alpha_1}_{1,i_1} \cdots \partial^{\alpha_n}_{n,i_n}(c_n) = c_0 \notin C'_0$

These operations naturally extend to labelled precubical sets.

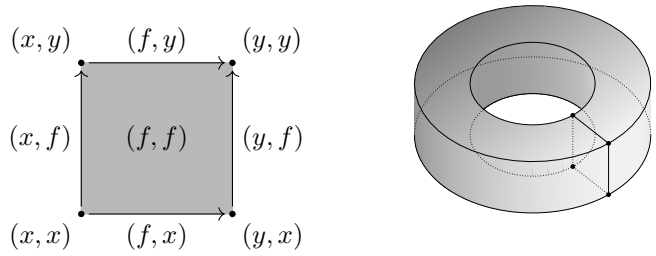
Example 1.3.11. We write I for the precubical set corresponding to the directed graph with a single edge f between two vertices x, y. Then $S^1 = I[x = y]$ is the precubical set with a single vertex and edge i.e. $\partial^{+}_{1,0}(f) = \partial^{+}_{1,0}(f) = x$.

$I =$ 

$S^1 =$ 

Then the tensor product $I \otimes I$ correspond to the filled square (which differs from the underlying graph by the 2-cube in grey). $I \otimes I \otimes S^1$ on the right below is the filled torus

where each arrow is a 1-cube, each face of the torus is a 2-cube and the interior is the only 3-cube.



$I \otimes I$ $\qquad$ $I \otimes I \otimes S^1$

Definition 1.3.12. A *path* π in a precubical set is a finite sequence $c_1, \dots, c_n$ of 1-cubes such that for every $1 \leq i < n$.

$$\partial^+_{1,0}(c_i) = \partial^-_{1,0}(c_{i+1})$$

Given two paths $\pi = (c_i)_{1 \leq i \leq n}$ and $\tau = (d_j)_{1 \leq j \leq p}$ such that $\partial^+_{1,0}(c_n) = \partial^-_{1,0}(d_p)$ we write $\pi \cdot \tau$ for their concatenation defined as the sequence $c_1 \dots c_n d_1 \dots d_p$

Definition 1.3.13. The *precubical transition set* associated to a command c is a precubical set G_c with labels in the set $\mathcal{L} = \mathcal{X} \coprod \mathcal{B}$, equipped with two distinguished 0-cubes $s_c, t_c \in E$ called the *start* and *end*. It is defined inductively as follows, similarly to Definition 1.2.2.

- $G_{\texttt{skip}}$ is the 0-cube.
- G_A for $A \in \mathcal{X}$ is precubical set with the 1-cubes labelled A with $\partial^-_{1,0}(A) = s_P$ and $\partial^+_{1,0}(A) = t_P$

$$s_A \bullet \xrightarrow{A} \bullet\ t_A$$

- $G_{c_1 ; c_2} = G_{c_1} \sqcup G_{c_2}[t_{c_1} = s_{c_2}]$

$$s_{c_1;c_2} = s_{c_1} \bullet \quad G_{c_1}\ t_{c_1} \bullet\ s_{c_2}\ G_{c_2} \quad \bullet\ t_{c_2} = t_{c_1;c_2}$$

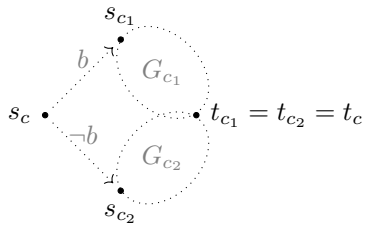
- With $c = \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2$,

$$G_c = (G_{c_1} \sqcup G_{c_2} \sqcup G_b \sqcup G_{\neg b})[s_b = s_{\neg b}, t_b = s_{c_1}, t_{\neg b} = s_{c_2}, t_{c_1} = t_{c_2}]$$

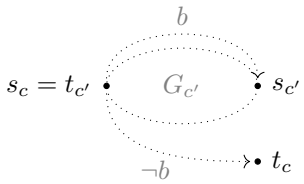
where

$$G_b = \quad s_b \bullet \xrightarrow{b} \bullet\ t_b \qquad G_{\neg b} = \quad s_{\neg b} \bullet \xrightarrow{\neg b} \bullet\ t_{\neg b}$$

Such that

$G_c =$ 

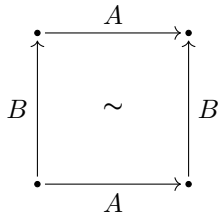
- With $c = \texttt{while}\ b\ \texttt{do}\ c'$, G_c is obtained similarly to the trnasition graph:



- $G_{c_1 \parallel c_2} = G_{c_1} \otimes G_{c_2}$. where $s_{c_1 \parallel c_2} = (s_{c_1}, s_{c_2})$ and $t_{c_1 \parallel c_2} = (t_{c_1}, t_{c_2})$

The pruned precubical transition set $\overline{\mathcal{G}}_c$ or *cubical semantics* of c is then obtained by restricting to valid vertices (i.e. 0-cubes) in the sense of Definition 1.2.25.

Example 1.3.14. Consider the program $P = A \parallel B$ where A and B are arbitrary actions. Its asynchronous semantics is shown below:

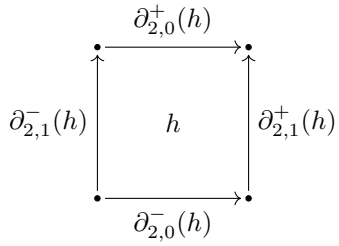


Most of the two cubes of the cubical semantics is obtained by adding a 2-cube to each square of this form in the transition graph of the program. Notice that there is a 2-cube introduced by the tensor product $G_A \otimes G_B$, even though the actions A and B might not commute.

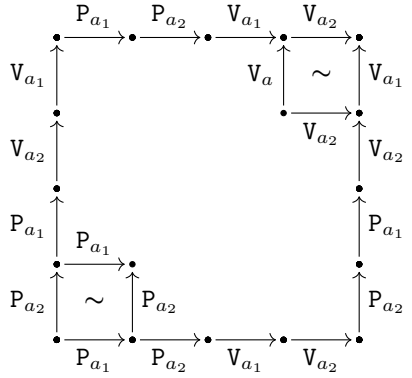
Definition 1.3.15. Given a precubical set C, the *dihomotopy* relation $\sim$ on paths is the smallest equivalence relation on path, which is a congruence w.r.t. concatenation and such that for all $h \in C_2$

$$(\partial^+_{2,1}(h) \cdot \partial^-_{2,0}(h)) \sim (\partial^+_{2,0}(h) \cdot \partial^-_{2,1}(h))$$

i.e. for any figure of the following form, the paths are dihomotopic.



Example 1.3.16. Let us consider the program $P = p_1 \| \dots \| p_n$ with $n+1$ resources $(a_i)_{0 \leq i \leq n}$, where $a_0 = a_n$ and $p_i = P_{a_i}; P_{a_{i+1}}; V_{a_i}; V_{a_{i+1}}$, commonly know as the "Dining Philosophers". For $n = 2$ its semantics is given below. For n philosophers there are more than 2^{2n} states and more than $2^{(n-1)^2}$ total traces. In comparison, there are only $2^n - 2$ maximal traces up to dihomotopy. Below we give the cubical semantics associated to the case of two philosophers:



Another example is the program $P = p_1 \| \dots \| p_n$ where $p_i = P_{a_i}; V_{a_i}$, where all a_i are distinct resources. There are $(2n)!$ maximal traces, but only a single dihomotopy class.

Since the dihomotopy relation is a congruence by definition, the following *fundamental category* can be associated to any precubical set.

Definition 1.3.17. The *fundamental category* $\vec{\Pi}_1(C)$ associated to a precubical set C is the category whose objects are the vertices of C and morphisms are paths up to dihomotopy.

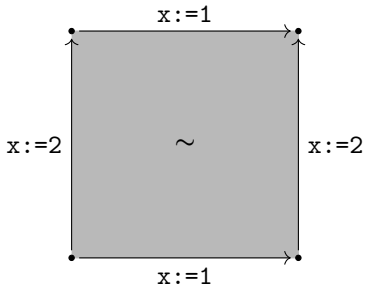
This gives us a more compact representation, of all possible paths up to dihomotopy and will prove useful in Section 1.3 in relating our models.

1.3.1.2 Coherent programs

Although it was the case in most of the examples up to now, nothing guarantees that the dihomotopy in the cubical semantics as defined in Definition 1.3.15 corresponds to the semantic one Definition 1.3.2, i.e. two paths are homotopic when they can be transformed into each other by permuting commuting actions as seen in Example 1.3.14. Programs for which this property holds are called coherent: in these programs, two dihomotopic paths have the same semantics.

Definition 1.3.18. A conservative program P is said to be *coherent* when for every pair of dihomotopic paths $\pi, \tau \colon x \to^* y$ in $\overline{\mathcal{G}}_P$, π is an execution trace if and only if τ is an execution trace. Furthermore, if this is the case, $[\![\pi]\!] = [\![\tau]\!]$.

Example 1.3.19. The program `x:=1‖x:=2` whose cubical semantics is given below is not coherent.



Indeed, the two paths $\pi = \texttt{x:=1}\cdot\texttt{x:=2}$ and $\tau = \texttt{x:=2}\cdot\texttt{x:=1}$ are related by a 2-cube and are thus dihomotopic, but their semantics are not the same $[\![\pi]\!](\sigma_0)(x) = 1 \neq 2 = [\![\tau]\!](\sigma_0)(x)$.

Remark 1.3.20. Every sequential program is coherent. Indeed, dihomotopic paths are generated by the constructor $\|$, which gets interpreted as a tensor product of precubical sets. In sequential programs, the dihomotopy relation is reduced to equality.

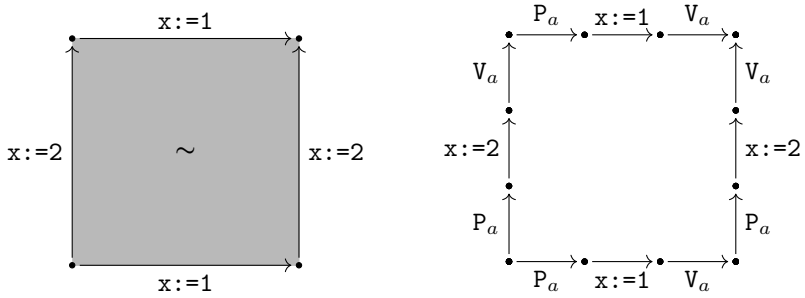
Coherent programs have a very nice property: two dihomotopic paths in a coherent program have the same effect on the state. Thus, in order to verify the executions of the program, it is enough to check a single representative for each dihomotopy class. This allows us to reduce the space of traces to explore during verification, giving results comparable to partial order reduction techniques [24, 52] which are for instance implemented in [32].

Even when compared up to dihomotopy, a program might generate an exponential number of maximal execution traces, for example the program $P = (\texttt{P}_a;\texttt{V}_a)\| \dots \|(\texttt{P}_a;\texttt{V}_a)$ of n parallel threads, whose cubical semantics is the skeletal cube of dimension n has $n!$ dihomotopy classes of execution traces.

In the following, we suppose that all programs are coherent. Following the POSIX philosophy, we leave to the programmer to make sure this assumption is verified. This is necessary as the property of being coherent is undecidable in general.

Remark 1.3.21. Given a program P, we can make it coherent by placing instructions $\texttt{P}_a$ and $\texttt{V}_a$, where a is a fresh mutex, around each action that is not locking or unlocking a resource as shown in the example below. This translation might seem naïve and particularly simple, but it is sometimes used in practical applications such as in the OCaml language [37]. Of course this translation is not optimal, and we refer to [18] for some eventual optimizations.

Example 1.3.22. Let us consider the program `x:=1‖x:=2`. It is not coherent since `x:=1` and `x:=2` do not commute. Its cubical semantics is shown on the left below and both maximal execution traces are dihomotopic.



The transformed program is $(\mathtt{P}_a;\mathtt{x:=1};\mathtt{V}_a)||(\mathtt{P}_a;\mathtt{x:=2};\mathtt{V}_a)$. Its pruned cubical semantics is shown on the right above. As no two paths are dihomotopic, it is coherent.

1.3.2 Geometric semantics

In this section we will present our final model of concurrent programs of our introductory chapter. This model is based on representing programs by topological spaces, equipped with a notion of direction, that we call *directed spaces*. The aim of this model is to import tools and techniques from algebraic topology in order to simplify the verification of programs. In those models, execution of a program is naturally assimilated to a path in the associated space. The reason we add a notion of direction is to encompass the causality of a program, the order in which operations must be executed (paths should not go back in time). This is why we define a notion of *directed paths*, a subset of all paths, respecting this causality. Many variants of this notion have been proposed, but we will mainly focus here on d-spaces as introduced by Grandis [25], as they provide more tractable results and are more widely accepted.

1.3.2.1 Directed spaces

Let us first recall some standard notions of classical topology:

- The unit interval I is the topological space $I = [0, 1]$ with the standard euclidean topology.
- A path f in a topological space is a continuous map $f\colon I \to X$. $f(0)$ (resp. $f(1)$) is called the source (resp. target) of f.
- We sometimes write $f\colon x \to y$ when talking about the path f when $x = f(0)$ and $y = f(1)$
- A loop is a path where $f(0) = f(1)$
- A path ε_x is constant when its image is a single point $\{x\}$

A directed topological space is then a topological space in the usual sense of the term, endowed with a coherent set of paths we call its directed paths

Definition 1.3.23. A *d-space* (X, dX) consists of a topological space X together with a set dX of paths of X - the *directed paths*, or *d-paths* of X - such that:

- All constant paths are in dX
- Given a directed path f and a weakly increasing function $\gamma\colon I \to I$ called a partial reparametrization, then $f \circ \gamma$ is also a directed path
- Given f, g two directed paths, such that $f(1) = g(0)$ the concatenation $f \cdot g$ defined as follows is also a directed path

$$f \cdot g(t) = \begin{cases} f(2t) & \text{if } 0 \leq t \leq 1/2 \\ g(2t-1) & \text{otherwise} \end{cases}$$

A subspace (Y, dY) of a d-space (X, dX) is a subset $Y \subseteq X$ with the topology inherited from X and where $dY = \{f \in dX \mid f(I) \subseteq Y\}$.

The second condition implies in particular that dX is closed under taking sub-paths.

Definition 1.3.24. A *morphism* of d-spaces, or *d-map*, $h\colon (X, dX) \to (Y, dY)$ is a continuous map $h\colon X \to Y$ such that for each $f \in dX$, $h \circ f \in dY$. We write **dTop** for the category of d-spaces and d-maps between them.

Example 1.3.25. The first example of a directed space is the *directed* unit interval $\vec{I} = (I, dI)$, with $I = [0, 1]$ the unit interval and dI is the set of weakly increasing paths of I. This structure is of course not unique, for example, we could have equipped I with the set of constant paths and still got a valid d-space.

Example 1.3.26. Many directed spaces can be generated from *pospaces*, that is, topological spaces X equipped with a partial order, whose graph is a closed subspace of $X \times X$ i.e. that is compatible with the topology of the space. The set dX for such a d-space will simply be the subset of weakly increasing paths. The directed interval from Example 1.3.25 is an example of such a d-space. Another example is the *directed n-cube* $\vec{I}^n$, generated by I^n, equipped with the product order.

An important property of **dTop** is that it has all limits and colimits [25, Section 1.4.1]. Furthermore, these should behave somewhat similarly to the (co)limits in the underlying category **Top** as shown by the following proposition.

Proposition 1.3.27. *The category* **dTop** *is both complete and co-complete. It has all limits and colimits. Furthermore, the forgetful functor* $U\colon$ **dTop** $\to$ **Top** *has both a left and a right adjoint. Thus, it preserves both limits and colimits.*

Proof. See [18, Proposition 4.5]. □

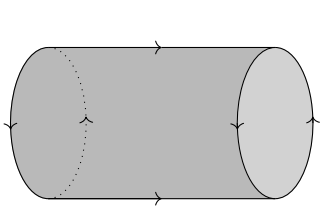
This allows us to give a concrete definition of some (co)limits on directed space, which will be useful constructors for the topological semantics.

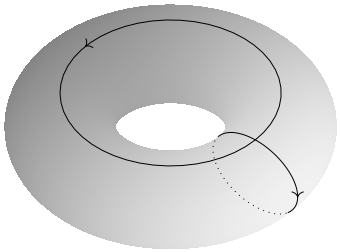
Definition 1.3.28. By Proposition 1.3.27, we can construct some usual (co)limits, as topological analogues of the operations on graph, as follows:

- The terminal d-space **1** is the containing a single point $\star$.
- The cartesian product $X \times Y$ of two d-space X and Y is defined as $(X \times Y, dX \times dY)$
- The disjoint union $X \sqcup Y$ of two d-space X, Y is defined as $(X \sqcup Y, dX \sqcup dY)$

- The quotient $X[x = y]$ of a d-space X is defined as $(X[x = y], dX[x = y])$, where $X[x = y]$ is the space obtained by identifying x and y and $dX[x = y]$ is the set (up to reparametrization) of finite sequence of directed paths $(f_i \colon s_i \to t_i)_{0 \leq i \leq n}$ in X such that $s_i, t_i \in \{x, y\}$, except possibly for s_1 and t_n.

Example 1.3.29. The product $\vec{S}^1 \times \vec{I}$ of the directed circle and the directed interval gives the filled cylinder on the left. The product $\vec{S}^1 \times \vec{S}^1$ gives the empty torus on the right. The product $\vec{S}^1 \times \vec{I} \times \vec{I}$ would give a filled square toroid similarly to Example 1.3.11.





1.3.2.2 Geometric semantics

The constructions of Section 1.3.2.1 allow us to define semantics of programs in directed topological spaces. These constructions are extremely similar to the constructions of the Definition 1.3.30 and Definition 1.2.2, adding a function to keep track of the consumption of resources.

Definition 1.3.30. Given an operational semantics, to any conservative program P, we associate a quadruple $(G_P, s_P, t_P, [\![]\!]_P)$ consisting of a directed topological space G_P equipped with two distinguished points $s_P, t_P \in G_P$, the beginning and the end, as well as a function $[\![-]\!]_P : G_P \to (\mathcal{R} \to \mathbb{Z})$, the *resource consumption*. This quadruplet is defined inductively as follows:

- For `skip`, writing $\mathbf{1} = \{\star\}$ for the terminal d-space:

$$G_{\texttt{skip}} = \mathbf{1} \qquad s_{\texttt{skip}} = \star \qquad t_{\texttt{skip}} = \star \qquad [\![x]\!] = \underline{0}$$

- For the locking $\texttt{P}_a$

$$G_{\texttt{P}_a} = \vec{I} \qquad s_{\texttt{P}_a} = 0 \qquad t_{\texttt{P}_a} = 1 \qquad [\![x]\!](b) = \mathbb{1}_{x > 0.5} \mathbb{1}_{b=a}$$

- For the unlocking $\texttt{V}_a$

$$G_{\texttt{V}_a} = \vec{I} \qquad s_{\texttt{V}_a} = 0 \qquad t_{\texttt{V}_a} = 1 \qquad [\![x]\!](b) = -\mathbb{1}_{x \geq 0.5} \mathbb{1}_{b=a}$$

- For any other action $A \in \mathcal{X}$,

$$G_A = \vec{I} \qquad s_A = 0 \qquad t_A = 1 \qquad [\![x]\!] = \underline{0}$$

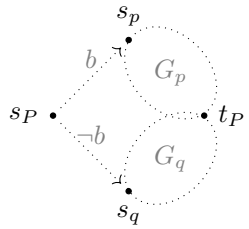
- $P = p; q$.

$$G_{p;q} = (G_p \sqcup G_q)[t_p = s_q] \qquad s_{p;q} = s_p \qquad t_{p;q} = t_q$$

$$[\![x]\!]_{p;q}(a) = \begin{cases} [\![x]\!]_p(a) & \text{if } x \in G_p \\ [\![x]\!]_p(t_p) + [\![x]\!]_q(a) & \text{if } x \in G_q \end{cases}$$

- $P = \texttt{if } b \texttt{ then } p \texttt{ else } q$

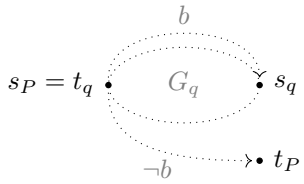
$$G_P = (G_p \sqcup G_q \sqcup G_b \sqcup G_{\neg b})[s_b = s_{\neg b}, t_b = s_p, t_{\neg b} = s_q, t_p = t_q]$$



$$s_P = s_b \qquad t_P = t_p \qquad [\![x]\!]_P(a) = \begin{cases} 0 & \text{if } x \in G_b \sqcup G_{\neg b} \\ [\![x]\!]_p(a) & \text{if } x \in G_p \\ [\![x]\!]_q(a) & \text{if } x \in G_q \end{cases}$$

- $P = \texttt{while } b \texttt{ do } q$.

$$G_P = (G_q \sqcup G_b \sqcup G_{\neg b})[s_q = t_b, t_q = s_{\neg b}, s_b = t_q]$$



$$s_P = s_b \qquad t_P = t_{\neg b} \qquad [\![x]\!]_P(a) = \begin{cases} 0 & \text{if } x \in G_b \sqcup G_{\neg b} \\ [\![x]\!]_q(a) & \text{if } x \in G_q \end{cases}$$

- $P = p \,\|\, q$

$$G_{p\,\|\,q} = G_p \times G_q \qquad s_{p\,\|\,q} = (s_p, s_q) \qquad t_{p\,\|\,q=(t_p,t_q)} \qquad [\![(x,y)]\!]_{p\,\|\,q} = [\![x]\!]_p + [\![y]\!]_q$$

Where given a condition b we have $G_b = (\vec{I}, 0, 1, \underline{0})$. The *forbidden region* is defined as the subspace

$$R_p = \{x \in G_P \mid \exists a \in \mathcal{R}, [\![x]\!](a) > \kappa_a \text{ or } [\![x]\!](a) < 0\}$$

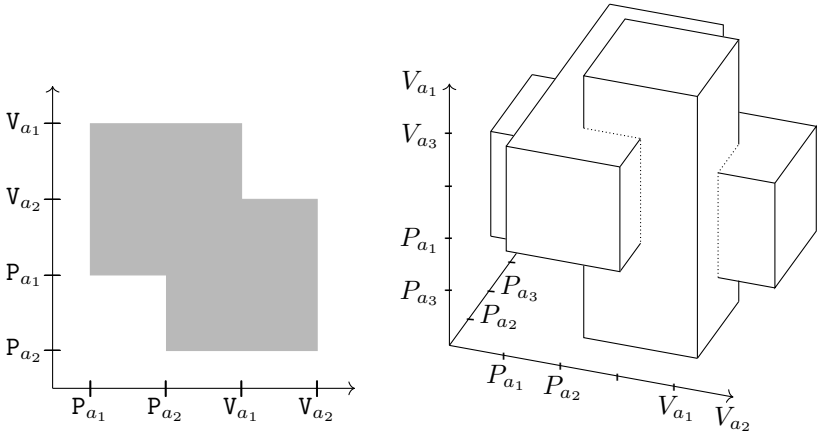
The *geometric semantics* $\overline{\mathcal{G}}_P$ of a program P is the d-space defined as $\overline{\mathcal{G}}_P = G_P \setminus R_P$

Remark 1.3.31. The resource consumption is well-defined only because we suppose that we are working with conservative programs.

The geometric semantics of a program can be seen as a "continuous" counterpart to the semantics we have presented up to now. To each program, we associate a d-space whose directed paths are the executions of a program.

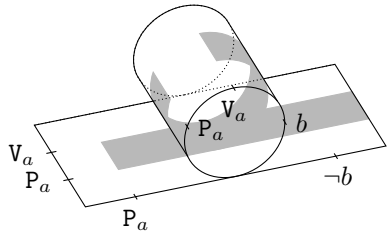
The commutation of the actions will be transposed as the equivalence of path up to continuous deformation, a version of "directed" homotopy, or *dihomotopy*. And we will prove that this notion of dihomotopy is actually equivalent to the one introduced in Definition 1.3.15.

Example 1.3.32. The geometric semantics of the dining philosophers from Example 1.3.16 is given below in dimension 2 and 3 respectively on the left and right.



Example 1.3.33. Below we give an example of the geometric semantics for the slightly less intuitive case of a program with loops

$$P = (\mathtt{P}_a;\ \mathtt{while}\ b\ \mathtt{do}\ (\mathtt{V}_a;\mathtt{P}_a))\|(\mathtt{P}_a;\mathtt{V}_a)$$



1.3.3 Homotopy in directed algebraic topology

1.3.3.1 Classical homotopy theory

The notion of homotopy is central to algebraic topology. Two maps with the same (co)domain are said to be homotopic when they can be continuously deformed into one another. We simply introduce basic notions to better introduce the study of dihomotopy, homotopy for directed spaces in the following section. We refer to the standard textbooks [4, 27] for more details.

Definition 1.3.34. Given two continuous maps between topological spaces $f, g\colon X \to Y$, a *homotopy* from f to g is a continuous map $h\colon I \times X \to Y$ such that $h(0,-) = f$ and $h(1,-) = g$. When such an h exists, the maps f and g are said to be homotopic, written $f \sim g$.

Remark 1.3.35. If $X = I$, then $f, g\colon I \to Y$ are paths on Y. And we can thus talk of homotopy of paths. Such a homotopy is said to be *endpoint preserving* when $h(-,0)$ is the constant function $\underline{f(0)}$ and $h(-,1) = \underline{f(1)}$.

In the following homotopy between two paths is always assumed to be endpoint-preserving.

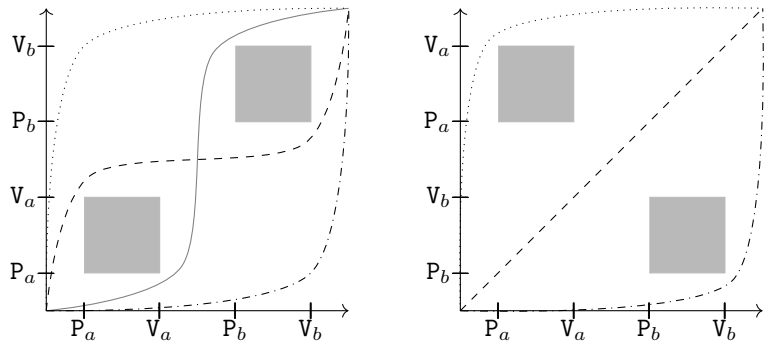
Example 1.3.36. Let us consider $X = I \times I$. Then for any two paths $f, g\colon x \to y$, if we define $h(t,x) = (1-t)f(x) + tg(x)$, then $h\colon I \times I \to X$ is an endpoint-preserving homotopy between f and g

In directed spaces, the notion of endpoint preserving homotopy between directed paths still makes sense, and we could define homotopy classes directed paths, and an equivalent of the fundamental category associated to a d-space. However, Example 1.3.37 suggests that when only considering directed paths, the notion of homotopy is not the right one.

Example 1.3.37. We give the respective geometric realization of the programs

$$\mathtt{P}_a;\mathtt{V}_a;\mathtt{P}_b;\mathtt{V}_b \| \mathtt{P}_a;\mathtt{V}_a;\mathtt{P}_b;\mathtt{V}_b \quad \text{and} \quad \mathtt{P}_a;\mathtt{V}_a;\mathtt{P}_b;\mathtt{V}_b \| \mathtt{P}_b;\mathtt{V}_b;\mathtt{P}_a;\mathtt{V}_a$$

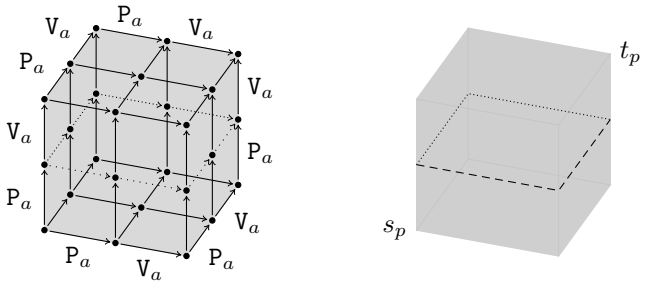
We consider all directed paths which are homotopic. This gives us on the left 4 homotopy classes, and on the right only 3 different paths up to homotopy. A representative from each class has been given on the semantics.



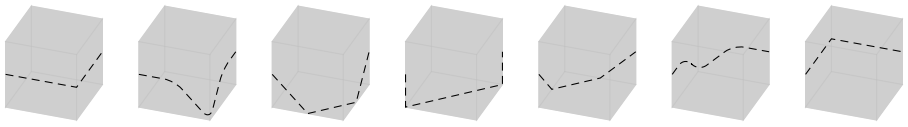
However, both underlying spaces are homeomorphic, and this implies that the homotopy types are similar, i.e. we should have the same homotopy classes.

1.3.3.2 Homotopy in directed spaces

Let us consider the program $P = \mathtt{P}_a;\mathtt{V}_a \| \mathtt{P}_a;\mathtt{V}_a \| \mathtt{P}_a;\mathtt{V}_a$. Its cubical semantics is the empty cube, with each cube of dimension 2 "filled", represented on the left of the figure below, while its geometric semantics is represented on the right.



Now let us consider the two paths highlighted in the cubical and geometric semantics. In the cubical semantics, they are not dihomotopic, whereas in the geometric semantics there is a homotopy between both paths as shown in the following figures.



Some paths used in the homotopy are not directed. And it would not be difficult to prove that it is not possible to build a homotopy between the paths that would only go through directed paths. In order for the relation to match the dihomotopy relation of cubical semantics, we will need a *directed* version of homotopy, that would be restricted to directed paths, preserving the idea that we cannot go back in time during execution.

Definition 1.3.38. Given a d-space X and two directed paths $f, g \colon \vec{I} \to X$, a *dihomotopy* between the two directed path f, g is an endpoint-preserving homotopy $h \colon I \times \vec{I} \to X$ from f to g such that for every $t \in I$, the path $h(t, -)$ is directed. In this case, f and g are called *dihomotopic*, written $f \sim g$.

Remark 1.3.39. The set of directed paths dX of a d-space X is in bijection with d-maps from $\vec{I}$ to X i.e. $dX \approx \mathbf{dTop}(\vec{I}, X)$ similarly to how the set of (non-directed) maps in X is by definition $\mathbf{Top}(I, X)$. Thus, f, g in the definition above are indeed directed paths of X.

Proposition 1.3.40. *The dihomotopy relation $\sim$ satisfies the following properties:*

- *The relation $\sim$ is an equivalence relation*
- *The relation $\sim$ is compatible with concatenation of paths. Given $f, f' \colon x \to y$ and $g, g' \colon y \to z$:*
$$f \sim f' \quad g \sim g' \quad \textit{implies} \quad f \cdot g \sim f' \cdot g'$$
- *Up to dihomotopy, concatenation of paths is associative and admits the empty path ε as identity. Given directed paths $f \colon w \to x$, $g \colon x \to y$ and $h \colon y \to z$ we have*
$$f \cdot (g \cdot h) \sim (f \cdot g) \cdot h \qquad \textit{and} \qquad \varepsilon_x \cdot f \sim f \cdot \varepsilon_y$$

Proof. See [18, Lemma 4.32] □

We can define the category of directed paths up to dihomotopy in a given topological space. Indeed, Proposition 1.3.40 guarantees that composition (concatenation) is well-defined and that the categorical axioms are well-defined.

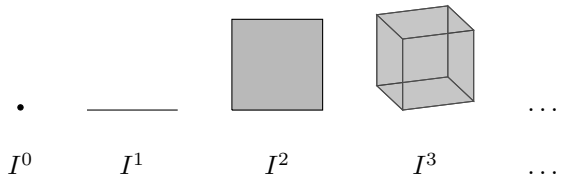
Definition 1.3.41. The *fundamental category* $\vec{\Pi}_1(X)$ of a directed space X is the category whose objects are the points of X and whose morphisms are the directed paths up to dihomotopy.

Remark 1.3.42. The same applies to the homotopy relation. Instead of having a category, we get a groupoid as the lack of direction allows to define an inverse for each path.

1.3.3.3 Geometric realization

In this section, we will relate our geometric model with our precubical model presented in Section 1.3.1 by showing how these precubical sets can be seen as topological spaces obtained by gluing cubes according to the data in the face maps.

Definition 1.3.43. The topological space I^n is called the *standard n-cube*.



As seen in the Example 1.3.26, we have a similar notion for directed spaces, and we'll call $\vec{I}^n$, called the *standard directed n-cube*, whose underlying space is I^n and directed paths are the weakly increasing maps.

Each standard n-cube has $2n$ faces, like n-cubes of precubical sets, a front and a back face in each direction, which are $(n-1)$-cubes included in $\vec{I}^n$ that can be described by the following inclusion maps:

$$\iota^-_{n,i} \colon I^{n-1} \to I^n \qquad\qquad \iota^+_{n,i} \colon I^{n-1} \to I^n$$

such that

$$\iota^\alpha_{n,i}(x_0, \dots, x_{n-1}) = (x_0, \dots, x_{i-1}, \mathbb{1}_{\alpha=+}, x_i, \dots, x_{n-1})$$

These face maps extend to directed standard n-cubes. It is thus natural to think of these (directed) standard n-cube as a (directed) topological counterpart of an n-cube. Then, a precubical set is simply a gluing of such cubes.

Formally, this means there is a functor $\vec{I}^- \colon \square \to \mathbf{dTop}$ associating to each element n a standard directed n-cube $\vec{I}^n$, that can be extended to a functor $|-| \colon \widehat{\square} \to \mathbf{dTop}$ defined as follows:

Definition 1.3.44. The *directed geometric realization* functor $|-| \colon \widehat{\square} \to \mathbf{dTop}$ is the functor associating a precubical set C to

$$|C| = \coprod_{n \in \mathbb{N}} (C_n \times \vec{I}^n) / \approx$$

Where

- C_n is equipped with the discrete topology and the discrete d-space structure (i.e. only constant paths are directed).
- $\approx$ is the equivalence relation generated by the relations $(\partial^{\alpha}_{n,i}(x), y) \approx (x, i^{\alpha}_{n-1,i}(y))$ for $n \in \mathbb{N}, x \in C_n$ and $y \in \vec{I}^{n-1}$.

Every 0-cube $x \in C_0$ is canonically associated to a point in $|C|$, denoted by $|x|$, elements of C_1 to segments... For further theoretical background on geometric realization we refer to [39, 22].

Example 1.3.45. The geometric realization of the precubical set corresponding to the cylinder and the torus given in Example 1.3.11 are the directed cylinder and directed torus described in Example 1.3.29.

This directed geometrical realization functor gives us a nice way of embedding precubical sets e.g. cubical semantics, in directed topological spaces, e.g. geometric semantic.

Proposition 1.3.46. *The directed geometric realization functor $|-|$ preserves all colimits and sends tensor products of finite precubical sets to cartesians products of their directed geometric realization.*

Proof. See [18, Proposition 4.12]. □

Proposition 1.3.47. *Given a conservative program P such that for each vertex x of the associated precubical set C_p, $[\![x]\!](a) \geq 0$ for all resources a. Then, the directed geometric realization of its cubical semantics $\overline{\mathcal{C}}_P$ embeds as a topological space in $\overline{\mathcal{G}}_P$:*

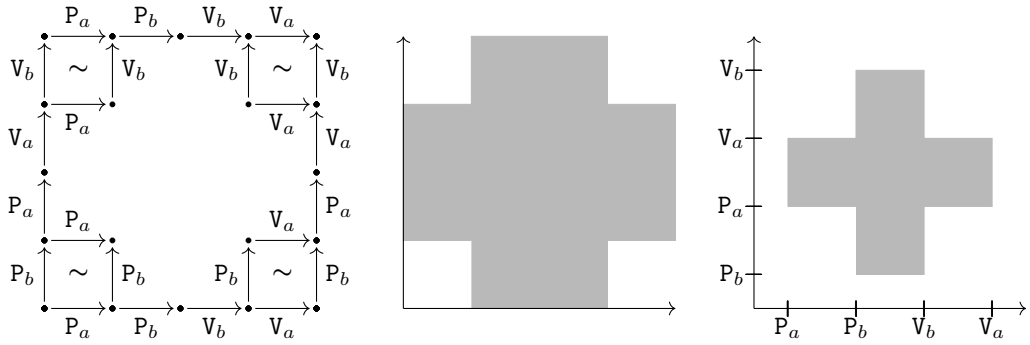
$$|\overline{\mathcal{C}}_P| \hookrightarrow \overline{\mathcal{G}}_P$$

Proof. See [18, Proposition 4.19]. □

Remark 1.3.48. The condition $[\![x]\!](a) \geq 0$ implies that no resources is released more than its capacity. This is in general good practice and can be considered to be enforced in practical cases.

Remark 1.3.49. By slightly modifying the definitions, the above embedding could be turned into an isomorphism, but would make it harder to understand. This can be safely assumed for the rest of the thesis.

Example 1.3.50. Consider the program $P = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$. From left to right we show its cubical semantics, its directed geometric realization and its geometric semantics: the embedding of the realization into the geometric semantics is trivial.



Thus, by Proposition 1.3.47, the geometric semantics $\overline{\mathcal{G}}_P$ corresponds to the cubical semantics $\overline{\mathcal{C}}_P$ via its directed geometric realization. Formally,

$$|\overline{\mathcal{C}}_P| = \overline{\mathcal{G}}_P$$

In particular, we can associate to each vertex $x \in \overline{\mathcal{C}}_P$ a point $|x| \in \overline{\mathcal{G}}_P$. We will now show that both the geometric and cubical semantics model the execution of programs in the same way. That is to say, they have the same directed paths. This correspondence on paths will only hold up to dihomotopy (as they are much more paths on $\overline{\mathcal{G}}_P$ than $\overline{\mathcal{C}}_P$). This would not be true should we consider paths simply up to homotopy.

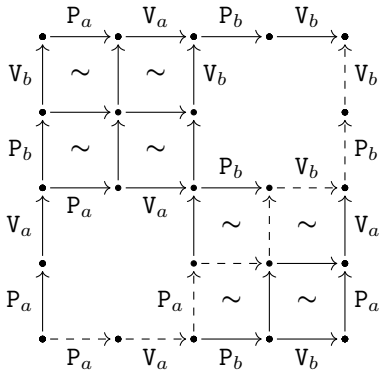
Theorem 1.3.51. *Given a conservative program P with $\overline{\mathcal{C}}_P$ (resp. $\overline{\mathcal{G}}_P$) as its cubical (resp. geometric) semantics. The functor induced by the directed geometric realization between the fundamental category of $\overline{\mathcal{C}}_P$ (Definition 1.3.17) and that of $\overline{\mathcal{G}}_P$ (Definition 1.3.41) is full and faithful:*

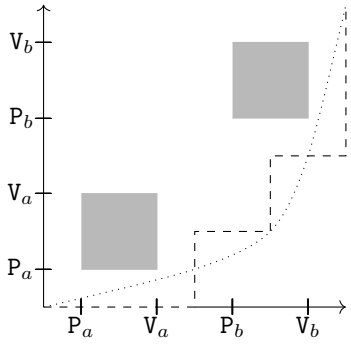
$$\vec{\Pi}_1(\overline{\mathcal{C}}_P) \hookrightarrow \vec{\Pi}_1(\overline{\mathcal{G}}_P)$$

This is equivalent to saying that for each vertex $x, y \in \overline{\mathcal{C}}_P$, there is a bijection between paths from x to y (in the cubical sense) and paths from $|x|$ to $|y|$ in $\overline{\mathcal{G}}_P$ (up to dihomotopy).

Proof. See [18, Theorem 4.38]. □

Example 1.3.52. Let us consider the program $P = \mathtt{P}_a;\mathtt{V}_a;\mathtt{P}_b;\mathtt{V}_b \| \mathtt{P}_a;\mathtt{V}_a;\mathtt{P}_b;\mathtt{V}_b$. Its cubical and geometrical semantics is respectively shown on the left and the right below.



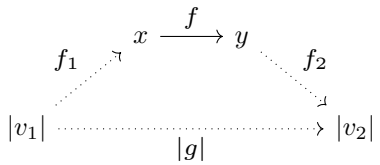


The dotted path in the geometric semantics is dihomotopic to the dashed path, which corresponds to the image of a path in the cubical semantics.

The embedding above cannot be directly extended to an equivalence of categories. Indeed, considering a portion of the dotted path in Example 1.3.52, no dihomotopic path which is the image of a cubical path can be found. However, it is always possible to extend a path in the geometric semantics so that it is dihomotopic to the realization of a cubical path.

Proposition 1.3.53. *Given a conservative program P, for every path $f\colon x \to y$ in $\overline{\mathcal{G}}_P$, there exists a path $g\colon v_1 \to v_2$ in $\overline{\mathcal{C}}_P$ and paths $f_1\colon |v_1| \to x$ and $f_2\colon y \to |v_2|$ such that*

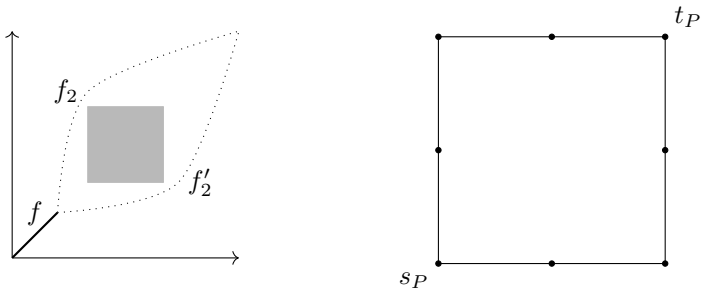
the paths $f_2 \cdot f \cdot f_1$ and $|g|$ are dihomotopic:



Moreover, if x (resp. y) is the realization of a point in $\overline{\mathcal{C}}_P$ then v_1 (resp. v_2) can always be chosen to be that point.

Proof. See [18, Proposition 4.40]. □

Remark 1.3.54. The extensions f_1 and f_2 provided in Proposition 1.3.53 are not canonical. For instance, consider the geometric semantics of the floating cube $P = \mathtt{P}_a;\mathtt{V}_a \| \mathtt{P}_a;\mathtt{V}_a$ depicted on the left. The path f can be extended in two different ways f_2 and f_2':



The path $f \cdot f_2$ and $f \cdot f_2'$ are respectively dihomotopic to each of the two maximal path of the geometric realization of $\overline{\mathcal{C}}_P$.

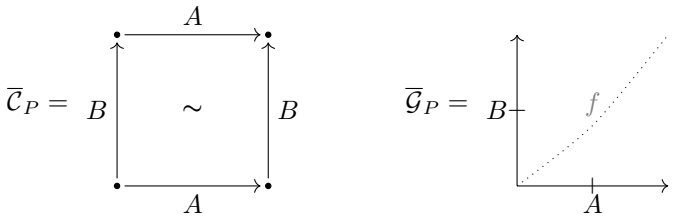
We can now define the geometric semantics of a path by reusing the semantics developed in Section 1.3.1 as follows.

Definition 1.3.55. Given a coherent conservative program P and two vertices $x, y \in \overline{\mathcal{C}}_P$, the *operational semantics* of a path $f\colon |x| \to |y|$ is the function $[\![f]\!]\colon \Sigma \to \Sigma$ defined as $[\![f]\!] = [\![g]\!]$ for some path $g\colon x \to y$ in $\overline{\mathcal{C}}_P$ such that $|g| = f$

The existence of g is guaranteed by Theorem 1.3.51. Furthermore, the definition does not depend on the choice of g. Indeed, two paths with the same realization are dihomotopic (in the cubical sense). As the program is coherent, their semantics are the same.

Remark 1.3.56. It can be shown that extremal points such as deadlocks and the endpoints of a program are the geometric realization of a vertex in $\overline{\mathcal{C}}_P$, so the definition covers most of our needs. We refer to [18] on how to extend this to other paths.

Remark 1.3.57. If we consider a program of the form $P = A \| B$. Its cubical and geometric semantics are respectively shown on the left and right below.



The path f is dihomotopic to the geometric realization of both maximal paths in the cubical semantics $B \cdot A$ and $A \cdot B$. Then we could define $[\![f]\!] = [\![A]\!] \circ [\![B]\!]$ or $[\![f]\!] = [\![B]\!] \circ [\![A]\!]$. If the program is not coherent, then there is no guarantee that $[\![A]\!] \circ [\![B]\!] = [\![B]\!] \circ [\![A]\!]$ and that the semantics of f can be defined.

The Proposition 1.3.53 allows us to consider the geometric counterparts of the "undesirable positions" introduced in Definition 1.1.29 and Proposition 1.2.35.

Definition 1.3.58. Given a conservative program P, the points x of its geometric semantics $\overline{\mathcal{G}}_P$ corresponding to Definition 1.1.29 can be identified as follows:

- If there is no directed path $f \colon s_p \to x$, then x is unreachable.
- If $x \neq t_P$ and the only directed path from x is the empty path ε_x, then x is a deadlock.
- If there exists a directed path $f \colon x \to y$, where y is a deadlock, then x is unsafe.
- If there is no dipath $f \colon x \to t_P$, then x is doomed.

The subspace of $\overline{\mathcal{G}}_P$ consisting of unreachable (resp. unsafe, resp. doomed) points is called the unreachable (resp. unsafe, resp. doomed) region.

Remark 1.3.59. We remind that these are only potential deadlock, unsafe, doomed points as they might be unreachable (in the sense that they might correspond to state that might not be reached by the program).

1.4 The boolean algebra of cubical regions

We present here the work of [18], which exhibits an algorithm for detecting deadlocks, unsafe and doomed positions in parallel programs. We will restrict our presentation to the cases of simple programs (Definition 1.4.1) and refer to [18] for the extension to parallel composition of more complex threads. The idea of this section is to represent subspaces of the geometric semantics $\overline{\mathcal{G}}_P$ by a set of "maximal" cubes that covers the underlying subspace. These sets of cubes give a finite representation with some nice properties of geometric realization of programs which is what we will be manipulating in the algorithms of Section 1.4.3 to find the deadlocks, unsafe and doomed regions.

1.4.1 Cubical cover of simple programs

Definition 1.4.1. A *simple program* is a program of the form $P = p_1 \| \dots \| p_n$, where the programs p_i consists purely of sequences of actions (i.e. they do not contain any conditional branching, loops or parallel composition). In this case, the programs p_i are called processes, and n is called the dimension of the program.

Simple programs are nice and easy to use as their geometric semantics correspond to a standard directed n-cube, with some cubes carved out as shown in Proposition 1.4.2.

Proposition 1.4.2. *The geometric semantics of a simple program P of dimension n is isomorphic to a d-space of the form*

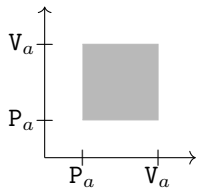
$$\overline{\mathcal{G}}_P = I^n \setminus \bigcup_{0 \leq i \leq k} R^i \qquad \text{with} \qquad R^i = \prod_{j=1}^{n}]x_j^i, y_j^i[$$

with $l \in \mathbb{N}$ and for all $i \in [1:n], x_j^i < y_j^i \in \{-\infty\} \cup \vec{I}^n \cup \{\infty\}$.

Proof. See [18, Equation (5.1)]. □

Furthermore, the regions R^i in Proposition 1.4.2 can be chosen to correspond to a conflict on a particular resource a_i. For a program with two threads and mutexes, this means that the lower bound of R^i will correspond to a thread trying to lock a_i while it is already locked up to its capacity. Similarly, the upper bound of R^i will correspond to an instruction V_{a_i} from both threads as in Example 1.4.3 below.

Example 1.4.3. Let us consider the program $P = \mathtt{P}_a; \mathtt{V}_a \| \mathtt{P}_a; \mathtt{V}_a$. Its geometric semantics, shown below is clearly isomorphic to $\vec{I}^2 \setminus]\frac{1}{3}, \frac{2}{3}[\times]\frac{1}{3}, \frac{2}{3}[$, where $]\frac{1}{3}, \frac{2}{3}[\times]\frac{1}{3}, \frac{2}{3}[$ clearly corresponds to the conflict on a. The precise coordinates of the forbidden region do not matter as we always consider up to isomorphism of d-space



For the rest of Section 1.4, we will only consider simple programs for ease of presentation, but the algorithms could be adapted to any programs without loops.

Definition 1.4.4. Given a space $X \subseteq \vec{I}^n$, a *cubical cover* $R = (R^i)_{i \in I}$ of X is a finite family of n-cubes with open or closed boundaries in X such that $\bigcup_{i \in I} R^i = X$.

Definition 1.4.5. A space X which admits a cubical cover is called a *cubical region*. We write $\mathcal{C}_n$ (resp. $\mathcal{R}_n$) for the set of cubical covers (resp. regions) included in $\vec{I}^n$

Remark 1.4.6. The results presented in [18] and in this section are for cubes that can have both open and closed boundaries. This has some effects on computations that will not be detailed here.

Finite cubical covers provide a nice way of representing the associated cubical region and manipulating them algorithmically. In the next Chapter 3, we will give a more precise implementation of these operations on cubes.

Proposition 1.4.7. *The set of cubical regions is closed under union, intersection and complement in $\vec{I}^n$. It is thus a boolean subalgebra of the power set $\mathfrak{P}(\vec{I}^n)$.*

Proof. See [18, Proposition 5.3]. □

1.4.2 Maximal cubical covers

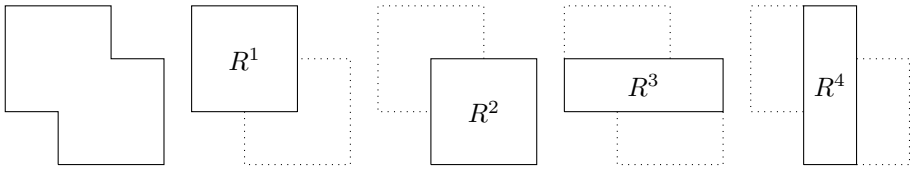
A cubical region generally admits multiple cubical covers, however there is always a canonical one which represents the cubical region, and usual operations can be performed quite efficiently on this representation.

Intuitively, a cover that uses bigger intervals is "more economical" than one that uses smaller intervals, as it requires less of those intervals to cover the same region. This gives us a first criterion for finding a most economical region, by ordering regions by inclusion of their intervals.

Definition 1.4.8. Let R and S be two cubical covers of a space X. The relation $\preceq$ defined as follows is a preorder on the cubical covers of X

$$R \preceq S \iff \forall R^i \in R, \exists S^j \in S, R^i \subseteq S^j$$

Example 1.4.9. Let us consider the space $X \subseteq \vec{I}^2$ on the left. Both $R = \{R^1, R^2, R^3, R^4\}$ and $S = \{R^1, R^2\}$ are cubical covers of X, but $S \preceq R$, as we consider that R carries more information about the region, as it has more of the maximal cubes included in X.



This is not enough to define the normal form as a maximal element. Indeed, taking from the previous Example 1.4.9, adding any cube included in R_1 to our cover gives an equivalent cover w.r.t. $\preceq$. We not only need covers that contain all maximal cubes, but we need a way of identifying covers that are more economical than others.

Definition 1.4.10. For the finite cubical covers of a space X, we can define a partial order $\leq$ from the preorder $\preceq$ as follows. Given R, S two cubical covers, we define

$$R \leq S \iff R \preceq S \preceq R \text{ and } S \subseteq R$$

It can be shown that the poset of cubical covers of X, equipped with $\leq$ defined above admits a maximum element. We will call this maximum element the *normal form* of the cubical region X, which consists of all the maximal n-cubes included in X. We will write $\mathcal{C}_n^{\max} \subseteq \mathcal{C}_n$ for the set of cubical covers in normal form.

Proposition 1.4.11. *Let us consider the functions $U_n \colon \mathcal{C}_n \to \mathcal{R}_n$ which associate to a cubical cover R the region $U_n(R) = \bigcup_{R^i \in R} R^i$ and $\mathcal{C}_n^{\max}$ which associate to a cubical region X, the finite set $\mathcal{C}_n^{\max}(X)$ of its maximal cubes w.r.t. to inclusion. Then there is an adjunction of functors, between the posets $(\mathcal{C}_n, \leq)$ and $(\mathcal{R}_n, \subseteq)$, called a Galois connection, written*

$$(\mathcal{C}_n, \leq) \underset{\mathcal{C}_n^{\max}}{\overset{U_n}{\rightleftarrows}} (\mathcal{R}_n, \subseteq)$$

When restricting to U_n to $\mathcal{C}_n^{\max}$, this Galois connection induces a bijection between the cubical regions $\mathcal{R}_n$ and the cubical covers in normal form $\mathcal{C}_n^{\max}$.

Proof. See [18, Proposition 5.6]. □

Among other things, this implies that $\mathcal{C}_n^{\max}(X)$ is the maximal cubical cover of a cubical region X w.r.t. to $\leq$, which we will define as the normal form of the cubical region X. The Galois connection above extends the boolean algebra structure to the normal forms, but it is not very useful in practice because we do not have yet a way of computing the normal cover associated to a region. But as we will see, we can compute "pre-normal" covers of the complement of cubical covers.

Definition 1.4.12. Given two n-cubes $R^1 = \prod_{k=1}^n [x_k^1, y_k^1]$ and $R^2 = \prod_{i=1}^n [x_k^2, y_k^2]$ of $\vec{I}$, we define their *intersection* $R^1 \cap R^2$ as follows:

$$R^1 \cap R^2 = \prod_{k=1}^{n} [\max(x_k^1, x_k^2), \min(y_k^1, y_k^2)]$$

Definition 1.4.13. Given two cubical covers $R = (R^i)_{i \in I}, S = (S^j)_{j \in J}$, we define their *intersection* $R \cap S$ as follows:

$$R \cap S = \{R^i \cap S^j \mid R^i \in R, S^j \in S\}$$

Proposition 1.4.14. *Given an n-cube $C = \prod_{j=1}^n]x_j, y_j[$. The maximal cubical cover of its complement is given by*

$$C^{\mathrm{c}} = \{\prod_{i=1}^{n} \mathbb{I}_{i,j}^0 \mid 0 \leq j \leq n\} \cup \{\prod_{i=1}^{n} \mathbb{I}_{i,j}^1 \mid 0 \leq j \leq n\}$$

where

$$\mathbb{I}_{i,j}^0 = \begin{cases} I & \text{if } i \neq j \\ [0, x_j] & \text{if } i = j \end{cases} \qquad \mathbb{I}_{i,j}^1 = \begin{cases} I & \text{if } i \neq j \\ [y_j, 1] & \text{if } i = j \end{cases}$$

Proof. It is trivial to check that all the cubes are in the complement. And by definition they can't be included in any other cube of the complement and any other cube will be included in such a cube. □

Remark 1.4.15. A similar property holds in the case where some boundaries of R are closed.

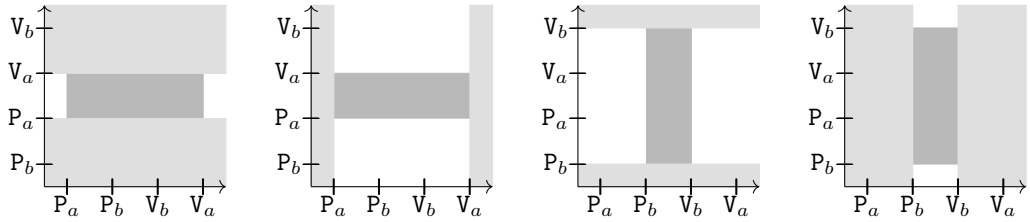
A cover is said to be *prenormal* if it contains the normal cover. This implies that we can obtain the normal cover by removing all the cubes that are included in another cube of the prenormal cover.

The intersection in Definition 1.4.13 of two prenormal region can be shown to still be prenormal. Then, a prenormal cover of the complement of any cubical cover $R = (R_i)_{i \in I}$ can be computed as the intersection of the normal (a fortiori prenormal) forms R_i^{c} of the complements of R_i of Proposition 1.4.14.

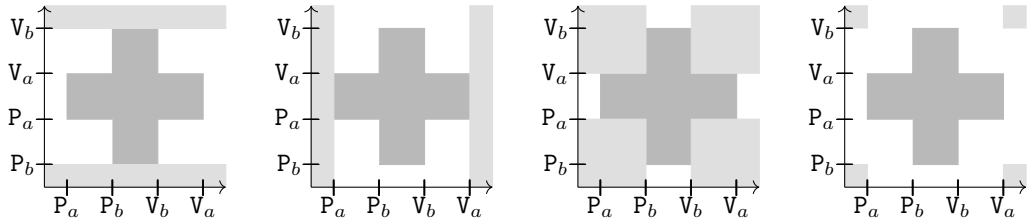
Proposition 1.4.16. *The complement of a cubical cover R of a cubical region X as given in Proposition 1.4.14 is a pre-normal cover of the region X^{c}.*

Proof. As discussed in the description of [18, Algorithm 5.7], it can be shown that intersection and complement of a prenormal cover are still prenormal. Thus, the complement of a cubical cover R computed as the intersection of the normal (a fortiori prenormal) forms R_i^{c} of the complements of R_i of Proposition 1.4.14 is prenormal. □

Example 1.4.17. Let us apply this to the Swiss Cross, whose semantics have already been described in Example 1.3.50. The forbidden region has two maximal cubes, whose complement are respectively shown on the two figures on the right and left.



By computing the intersections of all the cubes above in light grey, we get the following maximal cubes, as well as 4 unnecessary cubes shown on the right.



Thus, by taking the twice complemented region R^{cc}, and removing all the cubes that are not maximal, we get the normal form $\mathcal{C}_n^{\max}(X)$ of the cubical region X.

Computing the normal cover of X will allow us to compute the doomed and unsafe regions in Algorithm 1.4.33.

1.4.3 Computing deadlocks

1.4.3.1 Cubical partitions

The algorithm from [18, Section 5.1] works by first computing a suitable *cubical partition* of our allowed space X, then defining a relation between cubes of this partition corresponding to a notion of "directed connectedness".

Definition 1.4.18. Let $R = (R_i)_{i\in I}$ be a cubical cover of X, such that $\coprod_{i\in I} R_i = X$. Then we say that R is a *cubical partition* of X. We say that a partition R is coarser than a partition R' if $R \preccurlyeq R'$.

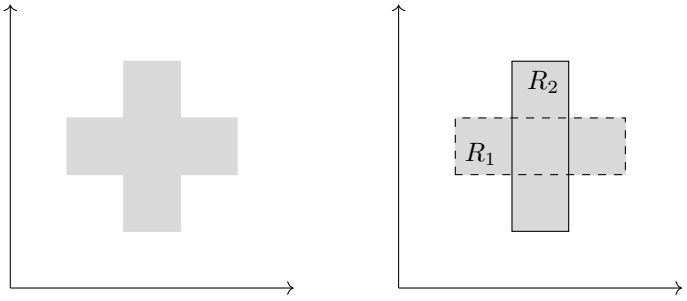
Definition 1.4.19. A partition R is *compatible* with a cubical cover C if for each $C_j \in C$, $R_i \subseteq C_j$ or $R_i \cap C_j = \emptyset$.

Let us suppose given a set X, and a subset R of $\mathfrak{P}(X)$, such that R covers X. The coarsest partition of X that is compatible with R is obtained by the following method:

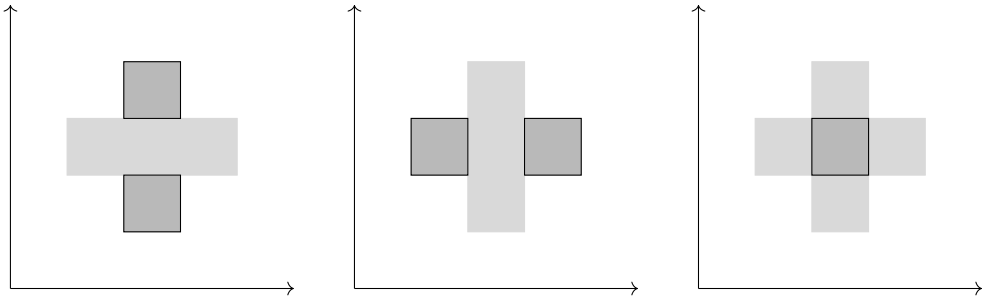
1. For each $R_i \in R$, we take either R_i or its complement R^{c}.
2. We perform the intersection of all the chosen elements.

For each different choice of R_i's, unless we chose the complement for each R_i, the intersection generates a subset of X. The set of all these intersections then generates the coarsest partition of X compatible with R.

Example 1.4.20. Let us consider the following subset of $[0,1] \times [0,1]$ in grey below, and the associated maximal cover:



The method described above gives us the three following subsets



$R_1^{\mathrm{c}} \cap R_2$ $\qquad$ $R_1 \cap R_2^{\mathrm{c}}$ $\qquad$ $R_1 \cap R_2$

In our case we are not actually dealing with a subset Y of $\mathfrak{P}(X)$, but with its associated cubical cover R. We would like the partition we obtain, to be a cubical cover, that is compatible with R and that it remains a partition when we consider the cubes of the cover. As we have explained that the underlying region already form a partition, the only thing to check is that we can find a maximal cubical cover of each element of the partition with no overlapping cubes.

We define a first try at a cubical cover compatible with a given partition

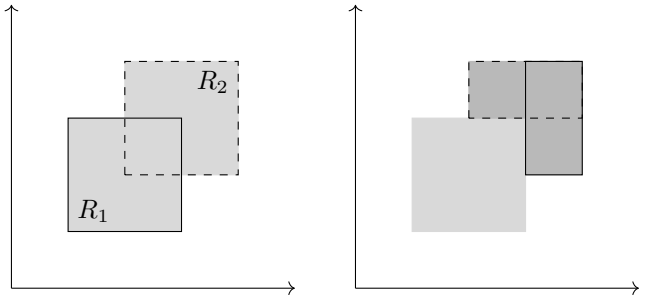
Definition 1.4.21. Given a cubical cover R of a set X, we define the *generic* 1*-dimensional cubical partition*:

$$\Gamma_1^m(R) = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} \mathcal{C}^{\max}(Z_{U,V})$$

with $Z_{U,V} = \bigcap_{u \in U} [u] \bigcap_{v \in V} [v]^{\mathrm{c}}$

Example 1.4.22. Let us consider the Example 1.4.20 above. Each of the subsets generated by the intersections does indeed admit a cubical cover, where none are overlapping.

Example 1.4.23. If the initial cover R of X was not maximal, this process does not give a proper cubical partition. Indeed, let us look at the region covered by $\{R_1, R_2\}$ below. The subsets of X generated by the partition do admit a cubical cover, but their cubes overlap.



Remark 1.4.24. It is important to specify that the cubical cover is maximal. Indeed, proving that a generic cubical cover with no overlapping exists is generally trivial, but harder to compute in practice. Or at least compute while keeping a reasonable size. This is important as the Algorithm 1.4.33 scales quadratically with the size of the cubical partition.

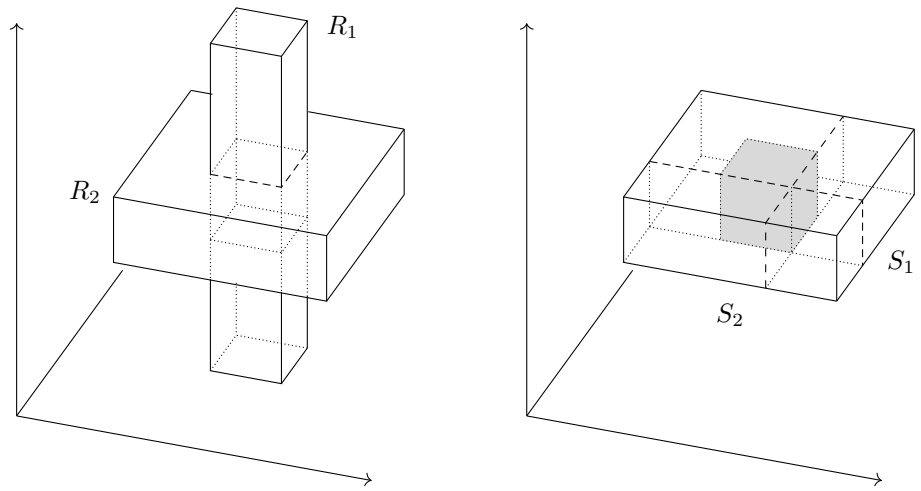
Unfortunately, this formula is not applicable beyond the dimension 2. Indeed, even for maximal cubical covers of cubical regions of $\vec{I}^3$, the method above does not give a cubical partition as shown in the Example 1.4.26 below.

Remark 1.4.25. In dimension 1, the method above does produce a cubical partition. Indeed, the complement of an interval $[x, y] \subseteq I$ is a region $[0, x[\times]y, 1]$, which is covered by two disconnected cubes. Then, the iterated intersection of cubes and their complements in dimension 1 will always lead to sets of disconnected cubes.

Example 1.4.26. Let us consider the program

$$P = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$$

where a, b are two semaphores of arity 2. The two cubes R_1, R_2 given below on the left form a cover of its forbidden region. Now, if we consider the intersection $R_1^{\mathrm{c}} \cap R_2$ on the right, we can see that the two maximal cubes S_1 and S_2 overlap in the lower right corner.



As we previously explained in Remark 1.4.24, it is important to find a partition as coarse as possible for efficiency, but actually computing the coarsest cubical partition is not necessary by any means, and any cubical partition will suffice.

For now, the best partition we can reasonably implement is the following "generic partition". Let us suppose given a subset $X \subseteq I^n$, and a cubical cover $R = (R_i)_{i \in I}$ of X. The *generic cubical partition* is obtained by the following steps:

1. For each axis $1 \leq k \leq n$, project each cube $R_i \in R$ on the axis k, i.e. for each cube $R_i = \prod_{j=1}^{n} [x_j^i, y_j^i]$, take the projection $[x_k^i, y_k^i]$.

2. For each axis $1 \leq k \leq n$, compute the coarsest cubical partition $\Gamma_1^m(([x_k^i, y_k^i])_{i \in I})$ associated to the cover $([x_k^i, y_k^i])_{i \in I}$

3. Compute cartesian product of all covers $\Gamma_1^m(([x_k^i, y_k^i])_{i \in I})$.

4. Remove all the cubes that are not included in X.

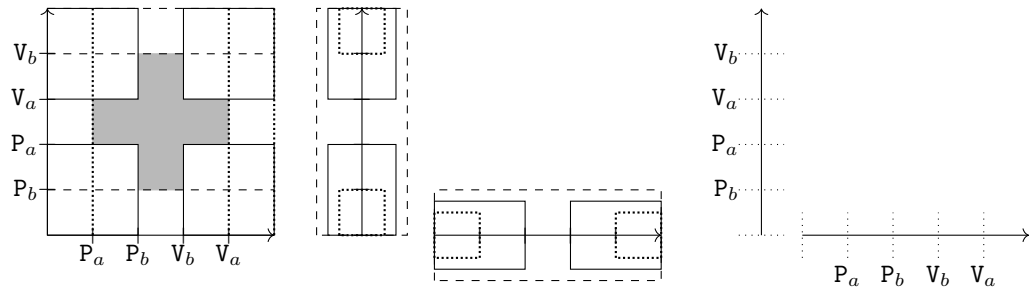
Definition 1.4.27. Given a cubical cover R of a region $X \subseteq \vec{I}^n$, we define the *generic cubical partition* $\Gamma^m(R)$ compatible with R as:

$$\Gamma^m(R) = \left(\prod_{i=1}^{n} \Gamma_1^m\big(\{c_i \mid \exists (r_k)_{1 \leq k \leq n} \in R, r_i = c_i\}\big) \right) \bigcap \mathcal{C}_n(R)$$

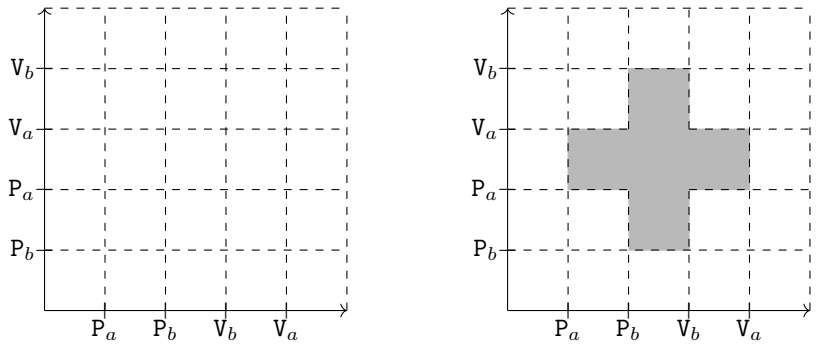
with Γ_1^m the generic 1-dimensional generic partition defined in Definition 1.4.21.

It is clear that this gives a partition compatible with the cover, and is what is used in [18, Chapter 5].

Example 1.4.28. Let us consider the Swiss Cross $P = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$, whose geometric semantics, and associated maximal cover R are given below on the left. Then, in the middle, we get the projection R^1 and R^2 of the cubes of R on both axes. Finally, on the right, the coarsest cubical partition $\Gamma_1^m(R^i)$, $i = 1, 2$ compatible with the projections.



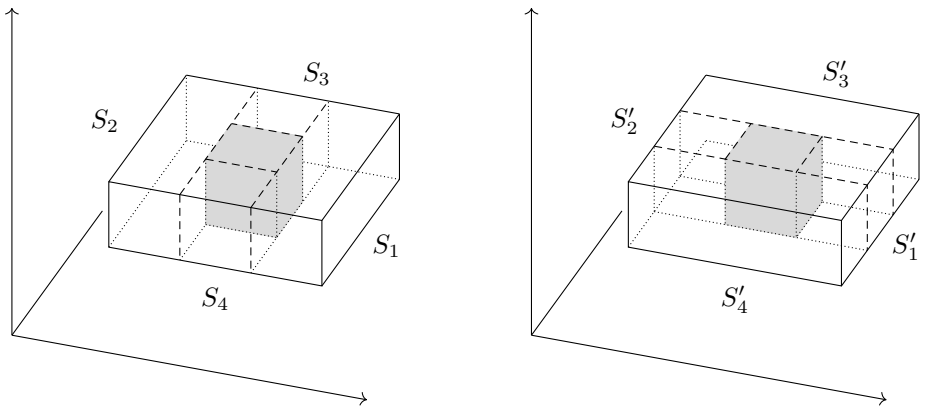
Computing the product $\Gamma_1^m(R^1) \times \Gamma_1^m(R^2)$, we obtain the partition on the left below. We see that this partition is not only a partition of the authorized region but of all $\vec{I}^2$. Then by removing the cubes whose associated cubical region is not in X, we get the partition of X on the right.



This method gives a cubical partition of the original subset X, which in practice, is quite close to the coarsest cubical partition. Of course some cubes might be partitioned too finely. One could think of trying to fuse cubes and see if the partition remains compatible. In practice the cost of the union and checking proves too great compared with simply executing Algorithm 1.4.33 on a suboptimal cover.

Furthermore, one can check that even for simple programs, the "coarsest cubical partition" might not be properly defined. In particular, going back to Example 1.4.26, there is no single maximal partition in terms of coarseness as shown below:

Example 1.4.29. Let us consider the Example 1.4.26, and more precisely, the region $R_1^c \cap R_2$, which is the square torus given below. In this case we have two maximal covers $S = (S_i)_{1 \leq i \leq 4}$ on the left and $S' = (S'_i)_{1 \leq i \leq 4}$ on the right. The two partitions given below are both local maximum in terms of coarseness.



Remark 1.4.30. As we have defined the intersection and complement of cubes and cubical covers, the computation can be performed directly on the cover for greater efficiency.

1.4.3.2 Ordering elements of the partition

The algorithm is based on a "reachability" order defined on the cubes of the coarsest partition. We say that a cube R_i is *in the past* of R_j, when every point of R_i is the origin of a directed path of X that reaches R_j. "Being in the past of" is actually the reflexive and transitive closure of the following relation.

Definition 1.4.31. Let $R = (R_i)_{i\in I}$ be a cubical partition of the space X. We define the pre-order $\lhd^*$ on elements of R as the reflexive transitive closure of the following relation: $R_i \lhd R_j$ if and only if for all $x \in [R_i]$ there exists $y \in [R_j]$ and a path of $[R_i] \cup [R_j]$ from x to y.

The relation $\lhd$ can be algorithmically computed by suitably comparing the boundaries.

Proposition 1.4.32. *Given a topological space $X \subseteq \vec{I}^n$, and two cubes R and S of X:*

$$R \lhd S \text{ if and only if } R \subseteq\downarrow ((\overline{R} \cap S) \cup (R \cap \overline{S}))$$

where, for any set Y, $\overline{Y}$ correspond to the topological closure of Y and $\downarrow Y = \{x \in X \mid \exists y \in Y, x \leq y\}$ is the downwards closure of Y.

Proof. [30, Proposition 7.5] □

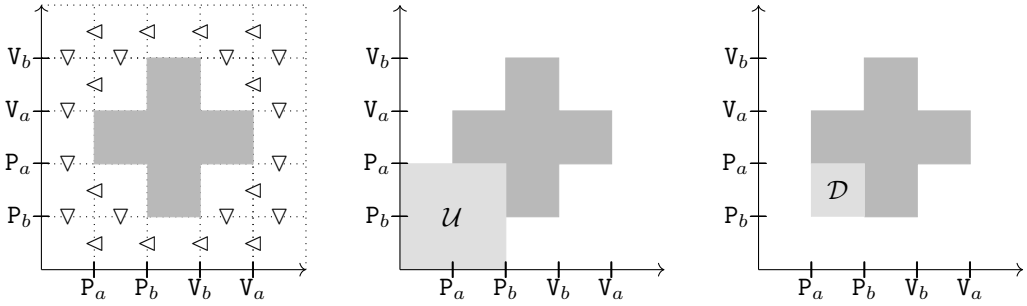
In the case of simple programs, the condition for being in the past is even stronger. Indeed, if $R \lhd S$, we can even ask the upper bound of R to be in the set $(\overline{R} \cap S) \cup (R \cap \overline{S})$ [30, Corollary 7.1]. Then it is only a matter of computing intersection of cubes (being careful around open borders).

It is on this resulting graph that we find the deadlocks using [18, Algorithm 5.7]. Indeed, the upper corner of an n-cubes R_i of this partition that is not the terminal cube of our program and such that there are no cubes R_j such that $R_i \lhd R_j$ is a deadlock [18, Section 5.2].

Algorithm 1.4.33. Let X^c be the forbidden region of a program P. $X \subseteq \vec{I}^n$. We can compute:

1. The normal cubical cover $\mathcal{C}_n^{\max}(X)$ of X as maximal cubes of the complement of X^{c}.
2. The generic cubical partition $\Gamma^m(\mathcal{C}_n^{\max}(X))$ that is compatible with $\mathcal{C}_n^{\max}(X)$.
3. Deadlocks are the cubes of $\Gamma^m(\mathcal{C}_n^{\max}(X)) \setminus R_{max}$ that are maximal (w.r.t. $\lhd^*$), where R_{max} is the only cube of $\Gamma^m(\mathcal{C}_n^{\max}(X))$ containing the maximal point of $\vec{I}^n$.
4. A cubical partition $\mathcal{U}(X)$ of the unsafe region as the downward closure (w.r.t. $\lhd^*$) of the deadlocks.
5. A cubical partition $\mathcal{D}(X)$ of the doomed region as $\mathcal{U}(X) \setminus \mathcal{E}(X)$, where $\mathcal{E}(X)$ is the downwards closure (w.r.t. $\lhd^*$) of R_{max}.

Example 1.4.34. Let us take the Swiss cross once more, $P = \mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a \| \mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b$ whose maximal cover has already been computed in Example 1.4.17. Then we can compute the coarsest partition of the allowed region, as shown on the left. From this we find the unsafe region $\mathcal{U}$ and doomed region $\mathcal{D}$.



This algorithm works well for programs without loops, but is quite complicated to implement (because of all the edge cases for the open and closed boundaries). Furthermore, this algorithm does not handle loops, although another algorithm from [18, Section 5.2] can handle some simple cases with some trade-offs. This will be explored later in Section 4.1.1.2. In Chapter 3, we will develop a new model of programs, based directly on the syntax of our programs, in order to make it easier to implement. We will show that even though we forewent the topological tools, we can still get back similar results, such as the deadlock detection Algorithm 1.4.33, with much more tractable results.

Chapter 2
Factoring models of programs

A coconut is just a nut.

– Oliver Lugg, *27 Unhelpful Facts About Category Theory*

One way of reducing the size of the state space of a concurrent program is to split it as several groups of processes running independently of each other, i.e. the execution of each group has no effect on the execution of the others. In such situations, we can analyse the processes independently and thus reason modularly on those. This process is very similar to the factorization of integers as products of prime factors or of polynomials as products of monomials.

In order to address this question, Ninin [46, 5] has provided an algorithm in order to factorize isothetic regions (as products of smaller regions), which covers in particular the geometric semantics of concurrent programs as introduced in Chapter 1. From this, we can deduce a factorization of the original program as a product of independent processes. In this chapter, we adopt another point of view and consider factorization of the *category of components* (instead of the geometric semantics). Informally, this category describes all the possible executions of the program, in a very compact way, by trivializing all transitions which have no real impact on the control flow of the program. For loop-free programs, such categories have the property of being *loop-free*, which roughly means that there is no non-trivial endomorphism. This motivates our quest for the more general question of factorizing loop-free categories.

In the 50s, Hashimoto has proved a very strong refinement result for posets [26, 49]: any two decompositions of a connected poset as a cartesian product of posets have a common refinement. In this chapter, we generalize this and show that this result in fact holds more generally for connected loop-free categories. This thus provides us with another approach to the decomposition of the semantics of concurrent (loop-free) programs.

We will first present the existing factorization techniques and framework introduced for program verification. Then we will motivate our extension of Hashimoto's theorem by introducing the category of components, a directed topological invariant which preserves products, and whose factorization leads to the factorization of programs. Finally, we will detail the proof of our refinement theorem for loop-free categories and the larger implications of this property for these categories.

2.1 Factorisation à la Ninin

2.1.1 Independent processes

Two process are independent when their execution can be freely interleaved without any effect on their evaluation, i.e. when the geometric semantics of their parallel composition is equal to the product of their geometric semantics:

Definition 2.1.1. In a program $P \| Q$, the processes P and Q are *independent* when

$$\overline{\mathcal{G}}_{P\|Q} = \overline{\mathcal{G}}_P \times \overline{\mathcal{G}}_Q.$$

Once this notion introduced, it is natural to investigate decompositions of programs as products of independent processes, which are maximal in the sense they have the greatest number of non-trivial factors. For some special classes of programs (parallel compositions of conservative programs), such decompositions have been shown to exist [5]. Similar decompositions have also been investigated in various related formalisms such as CCS [41] or π-calculus [16]. Such decompositions are useful because we can show they provide normal forms for programs (up to reordering of processes), similar to the decomposition of integers as products of prime factors.

In some cases, it is very simple to determine when two processes are independent. For instance, a simple criterion is given in [5, 31]: two processes are independent when the sets of resources occurring in each of the two processes are disjoint.

Example 2.1.2. Consider the following program P:

$$\begin{aligned} P = \quad & \mathtt{P}_a\,;\mathtt{V}_a \\ \| \quad & \mathtt{P}_b\,;\mathtt{V}_b \\ \| \quad & \mathtt{P}_b\,;\mathtt{V}_b \\ \| \quad & \mathtt{P}_a\,;\mathtt{V}_a \end{aligned}$$

It is easy to see that the program P can be factorized in two minimal concurrent programs

$$P = \Big(\mathtt{P}_b\,;\mathtt{V}_b \,\|\, \mathtt{P}_b\,;\mathtt{V}_b\Big) \,\|\, \Big(\mathtt{P}_a\,;\mathtt{V}_a \,\|\, \mathtt{P}_a\,;\mathtt{V}_a\Big)$$

Indeed, the two subprograms have no mutex in common (the first one uses only b and the second one only a).

Of course, the above condition is sufficient but not necessary, as illustrated in the following example.

Example 2.1.3. Fix two mutexes a, b and a semaphore c of arity 2, and consider the program

$$\begin{aligned} P = \quad & \mathtt{P}_a\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_a \\ \| \quad & \mathtt{P}_a\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_a \\ \| \quad & \mathtt{P}_b\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_b \\ \| \quad & \mathtt{P}_b\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_b \end{aligned}$$

This program can ba factorized as

$$P = \Big(\mathtt{P}_a\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_a \parallel \mathtt{P}_b\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_b\Big) \parallel \Big(\mathtt{P}_a\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_a \parallel \mathtt{P}_b\,;\mathtt{P}_c\,;\mathtt{V}_c\,;\mathtt{V}_b\Big)$$

since the mutexes a, b prevent c from being taken more than twice at the same time. We will study this example in more details below.

2.1.2 The free commutative monoid of cubical covers

The setting in which factorization properties are best studied is the one of free commutative monoids that we now recall.

Definition 2.1.4. A *monoid* $(M, \cdot, e)$ is a set M equipped with an internal multiplication $- \cdot -\colon M \times M \to M$ and an element e such that the following properties are satisfied:

- associativity: for any $x, y, z \in M$, $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
- unitarity: for any $x \in M$, $x \cdot e = e \cdot x = x$

We say that the monoid is *commutative* when $x \cdot y = y \cdot x$ for every $x, y \in M$.

Example 2.1.5. $(\mathbb{N}, +, 0)$ is a commutative monoid.

Example 2.1.6. $(\{e, a, b\}, \cdot, e)$ with $\cdot$ defined as follows is a non-commutative monoid.

$$\begin{aligned} a \cdot b &= a & \qquad b \cdot a &= b \\ a \cdot a &= a & \qquad b \cdot b &= b \end{aligned}$$

Definition 2.1.7. Let $(M, \cdot, e)$ a monoid. An element x of M is said to be *invertible* when there exists an element x^{-1} of M such that $x \cdot x^{-1} = x^{-1} \cdot x = e$. When such an element exists, it is unique and called the *inverse* of x.

Definition 2.1.8. Let $(M, \cdot, e)$ a monoid. A non-invertible element $x \in M$ is said to be *irreducible* if for all elements $y, z \in M$, $x = y \cdot z$ implies y or z is invertible.

We recall that a word w on an alphabet $\mathbb{A}$ is a finite sequence of elements of $\mathbb{A}$ and that we write $\mathbb{A}^*$ for the set of all words over $\mathbb{A}$. The concatenation of two words $w = w_1 \dots w_p$ and $w' = w'_1 \dots w'_q$ is given by $w \cdot w' = w_1 \dots w_p w'_1 \dots w'_q$.

Definition 2.1.9. Given a set S, the *free monoid* over the set S is the monoid $(S^*, \cdot, \varepsilon)$, where S^* is the set of all words over S, equipped with concatenation.

Definition 2.1.10. A monoid $(M, \cdot, e)$ is said to be *free* whenever it is isomorphic to the free monoid over the set M.

Proposition 2.1.11. *In a free commutative monoid, every element can be uniquely factored (up to permutation of terms) as a product of irreducible elements.*

Proof. [23, Theorem 1.2.9] □

Definition 2.1.12. Given a free commutative monoid $(M, \cdot, e)$, and an element m of M, the *prime factorization* is defined as the unique factorization of m as a product of irreducible element.

Example 2.1.13. The monoid $(\mathbb{N}^*, \times, 1)$ is a free monoid, with irreducible elements the prime numbers.

Example 2.1.14 (from [45]). Let $(\mathbb{N}[X], \times, 1)$ the commutative monoid of polynomials with coefficients in $\mathbb{N}$. It is not fre. Indeed the polynomial $\sum_{i=0}^{5} X^i$ can be decomposed in two different products of irreducible polynomials:

$$\sum_{i=0}^{5} X^i = (X^3+1)(X^2+X+1) = (X+1)(X^4+X^2+1)$$

The sets of *n-cubes* of the geometric semantics of a program as defined in Chapter 1 is a free commutative monoid when we consider it as the set of homogenous (of the same size) words over the alphabet $\mathbb{A}$ consisting of the set of all sub-intervals of $\vec{I}$ [18, 31]. Indeed, let $\mathbb{A}$ be the set of subintervals of $\vec{I}$, then an n-cube is precisely a word of size n on the alphabet $\mathbb{A}$. Furthermore, a cubical cover is a homogenous finite set of such words.

The concatenation operator Definition 1.1.20 induces a monoidal structure on cubical covers which is itself induced by the cartesian product (equivalent to parallel composition of processes).

Example 2.1.15. The cube $[0,1] \times [0,1] \times [0, \frac{1}{2}]$ and $[\frac{1}{2}, 1] \times [0,1]$ can be respectively seen as words of length 3 and 2 over the set of sub-intervals of $[0,1]$. Their concatenation is simply given by the usual cartesian product

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

Definition 2.1.16. The group $\mathfrak{S}_n$ of all permutations of the set $[1:n]$ is equipped with the tensor product $\otimes$ defined, for all $\sigma \in \mathfrak{S}_p$ and $\tau \in \mathfrak{S}_q$ by $\sigma \otimes \tau \in \mathfrak{S}_{p+q}$ where:

$$(\sigma \otimes \tau)(i) = \begin{cases} \sigma(i) & \text{if } i \leq p \\ \tau(i-p)+p & \text{if } p < i \leq p+q \end{cases}$$

We have an action of $\mathfrak{S}_n$ on the set of n-cubes is given on an n-cube c by

$$\sigma.c = (c_{\sigma^{-1}(1)}, \ldots, c_{\sigma^{-1}(n)})$$

This action extends to all homogenous sets of cubes (i.e. cubical covers) R,

$$\sigma.R \quad ::= \quad \{\sigma.c \mid c \in R\}$$

The tensor product is compatible with the cartesian product of topological spaces:

Lemma 2.1.17. *Given two spaces $X \subseteq \vec{I}^p$ and $Y \subseteq \vec{I}^q$, and two permutations $\sigma \in \mathfrak{S}_p, \tau \in \mathfrak{S}_q$ we have $(\sigma.X) \times (\tau.Y) = (\sigma \otimes \tau).(X \times Y)$.*

Proof. See [18, Lemma 5.23]. □

We also have the expected analogous compatibility property for regions. First we recall the definition of a quotient.

Definition 2.1.18. Given a group $\mathfrak{S}$ acting over a set X, we define the *quotient* $\mathcal{X}/\mathfrak{S}$ as the set of all orbits of elements of X under the action of $\mathfrak{S}$:

$$\mathcal{X}/\mathfrak{S} = \{\{\sigma.c \mid \sigma \in \mathfrak{S}_n\} \mid c \in X\}$$

We can then define the following monoids.

Definition 2.1.19. Let $\overline{\mathcal{G}}_p$ be the geometric semantics of a program P and $\mathcal{R}_n$ (resp. $\mathcal{C}_n$) be the set of all cubical regions (resp. cubical covers in normal forms) of n-cubes.

- Then $\mathcal{R}^{\mathfrak{S}} = \coprod_{n\in\mathbb{N}} \mathcal{R}_n/\mathfrak{S}_n$, equipped with the concatenation of Definition 1.1.20 (induced by the cartesian product) is the *monoid of cubical regions over* $\overline{\mathcal{G}}_p$ (up to permutation). Its neutral element is the non-empty 0-dimensional cubical region.
- Similarly, $\mathcal{C}^{\mathfrak{S}} = \coprod_{n\in\mathbb{N}} \mathcal{C}_n^{\max}/\mathfrak{S}_n$ is the *monoid of cubical covers in normal form* (up to permutation) of $\overline{\mathcal{G}}_p$.

Proposition 2.1.20. *The monoid $\mathcal{R}^{\mathfrak{S}}$ is a free commutative monoid which is isomorphic to $\mathcal{C}^{\mathfrak{S}}$.*

Proof. [18, Theorem 5.25] □

Given a program P, Proposition 2.1.20 and Proposition 2.1.11 imply that every cubical region, and in particular $\overline{\mathcal{G}}_P$ can be factored as a product of irreducible elements. Furthermore, the fact that $\mathcal{R}^{\mathfrak{S}}$ are isomorphic $\mathcal{C}^{\mathfrak{S}}$ allows us to perform our computation on the maximal cover of $\overline{\mathcal{G}}_P$ instead.

2.1.3 Factorization and partition

We have seen above that cubical covers can be factored uniquely as products of irreducible elements. To make a use of this result in practice, we need to be able to compute this factorization.

For implementation matters, it is convenient to represent such factorizations as partitions of $[1:n]$, which are linked to factorization of a homogenous set of elements of a monoid by Lemma 2.1.23 below.

Definition 2.1.21. Given an alphabet $\mathbb{A}$ and a word $w = w_1 \dots w_n \in \mathbb{A}^*$. Let I be a subset of $[1:n]$, we write $w|_I$ for the subword of w consisting of letters with indices in I. Given a set $R \subseteq \mathbb{A}^n$, we define $R|_I = \{w|_I \mid w \in R\}$.

Definition 2.1.22. Let $w = w_1 \dots w_n \in \mathbb{A}^n$, let $I \subseteq [1:n]$. We write

$$\begin{aligned} \pi_I : \mathbb{A}^n &\to \mathbb{A}^{n-|I|} \\ w &\mapsto w|_I \end{aligned}$$

for the canonical projection.

Lemma 2.1.23. *Given $I \subseteq [1:n]$, we write $I^{\mathrm{c}} = [1:n] \setminus I$. For a homogenous set $R \subseteq \mathbb{A}^n$ of cubes, we have $R = R|_I \times R|_{I^{\mathrm{c}}}$ if and only if for all $u, v \in R$ there exists $w \in R$ such that $w|_I = u|_I$ and $w|_{I^{\mathrm{c}}} = v|_{I^{\mathrm{c}}}$.*

Proof. See [18, Lemma 5.27]. □

In order to find the prime factorization (Definition 2.1.12) of a cubical cover $R \subseteq \mathbb{A}^*$ it is enough to check all the subsets I of $[1:n]$ of cardinality less than $n/2$ and, for each of those, check if there is a decomposition of R as $R|_I \times R|_{I^c}$ using the characterization given by the above lemma. This gives us the following algorithm, with dramatic complexity, which brutally checks all possible decomposition of a set of cubes R as a product of $R|_I \times R_{I^c}$ (up to permutation), for a given subset $I \subseteq [1:n]$.

Algorithm 2.1.24. Given $R \subseteq \mathbb{A}^n$ a cubical cover of the state space in normal form.

1. Choose a set $I \subseteq [1:n]$ such that its cardinal $p \leq n/2$.
2. Compute $S_I = \{\pi_{I^c}(\pi_I^{-1}(w)) \mid w \in \pi_I(R)\}$.
3. If S_I is a singleton then R factorizes as $R|_I \times R|_{I^c}$. Otherwise, go back to the first step and pick another I.
4. Proceed recursively with the same algorithm on $R|_I$ and $R|_{I^c}$.
5. Once all sets I are depleted, R is no longer factorizable.

If $I = [1:p]$, then the set $\pi_{I^c}(\pi_I^{-1}(w))$ associated to a prefix $w \in \pi_I(R)$ of size p, is the set of all suffixes of size $n-p$ of words starting by the prefix w. Then, the fact that S_I is a singleton correspond to the fact, that any concatenation $w \cdot w'$ of such a prefix and suffix is a cube of R.

Example 2.1.25. Let us consider the Example 2.1.3 once more. We remind ourselves that we have two mutexes a, b and a semaphore c of arity 2 and a program

$$\begin{aligned} P = \quad & \pi_1 = \mathtt{P}_a;\mathtt{P}_c;\mathtt{V}_c;\mathtt{V}_a \\ \| \quad & \pi_2 = \mathtt{P}_b;\mathtt{P}_c;\mathtt{V}_c;\mathtt{V}_b \\ \| \quad & \pi_3 = \mathtt{P}_a;\mathtt{P}_c;\mathtt{V}_c;\mathtt{V}_a \\ \| \quad & \pi_4 = \mathtt{P}_b;\mathtt{P}_c;\mathtt{V}_c;\mathtt{V}_b \end{aligned}$$

As seen in Example 2.1.3, even though the independence criterion is not respected, it is possible to factorize this program. The normal cubical cover R of the geometric semantics has the following 16 cubes, where we use $[0,5]^4$ instead of $\vec{I}^4$ for readability:

$$\begin{array}{ll} [0,5]\times[4,5]\times[0,1]\times[0,5] & [4,5]\times[0,5]\times[0,1]\times[0,5] \\ [0,5]\times[4,5]\times[0,5]\times[0,1] & [4,5]\times[0,5]\times[0,5]\times[0,1] \\ [0,5]\times[4,5]\times[0,5]\times[4,5] & [4,5]\times[0,5]\times[0,5]\times[4,5] \\ [0,5]\times[4,5]\times[4,5]\times[0,5] & [4,5]\times[0,5]\times[4,5]\times[0,5] \\ \\ [0,5]\times[0,1]\times[0,1]\times[0,5] & [0,1]\times[0,5]\times[0,1]\times[0,5] \\ [0,5]\times[0,1]\times[0,5]\times[0,1] & [0,1]\times[0,5]\times[0,5]\times[0,1] \\ [0,5]\times[0,1]\times[0,5]\times[4,5] & [0,1]\times[0,5]\times[0,5]\times[4,5] \\ [0,5]\times[0,1]\times[4,5]\times[0,5] & [0,1]\times[0,5]\times[4,5]\times[0,5] \end{array}$$

Now it is easy to see that independently of the first half of our cubes we consider, the set of associated suffixes will always be the following ones:

$$[0,1]\times[0,5] \qquad [0,5]\times[0,1] \qquad [0,5]\times[4,5] \qquad [4,5]\times[0,5]$$

This is precisely the condition $S_I = \{\pi_{I^c}(\pi_I^{-1}(w)) \mid w \in \pi_I(R)\}$ for $I = \{1,2\}$, i.e.

$$R = R|_{1,2} \times R|_{3,4}$$

And indeed, the subprograms $\pi_1 \| \pi_2$ and $\pi_3 \| \pi_4$ are independent. The algorithm, will then proceed recursively on subsets of $\{1,2\}$ and $\{3,4\}$ without finding suitable subsets as the program cannot be factorized further.

This algorithm is not limited to programs whose semantics are n-dimensional cubes, but can be extended to any simple program, i.e. any program of the form $P_1 \| P_2 \| \dots \| P_n$ where each P_i does not contain parallel composition, but might contain branchings and even loops [18].

2.1.3.1 A more efficient factorization algorithm

Algorithm 2.1.24 can be further improved, thanks to a result by Haucourt and Ninin [31], by exploiting the fact that the n-dimensional cubical regions form a boolean algebra.

Proposition 2.1.26. *Let $R \subseteq \mathbb{A}^n$ be the cubical cover in normal form of the complement of the state space $X \subseteq \vec{I}^n$. The prime factorization of X is*

$$X = \pi|_{I_1}(X) \times \dots \times \pi|_{I_n}(X)$$

such that $I_1, \dots, I_n$ is the finest partition of $[1:n]$ whose elements are unions of subsets of the form $\{i \mid \pi_i(u) \neq \vec{I}\}$.

Proof. See [46, Proposition 2.4.10]. □

We can then refine our Algorithm 2.1.24 to be used on the forbidden region which is generally much smaller.

Example 2.1.27. Let us restart Example 2.1.25. The forbidden region of the program is

$$\{]1,4[\times]1,4[\times[0,5]\times[0,5], [0,5]\times[0,5]\times]1,4[\times]1,4[\}$$

From Proposition 2.1.26, the factorization immediately follows. The associated partition of $[1:4]$ being $\{\{1,2\},\{3,4\}\}$.

2.2 The category of components

2.2.1 Category of components of loop-free programs

A major contribution of algebraic topology is to provide invariants of topological spaces up to homotopy, such as homotopy groups or homology groups. One of the simplest such invariants is the set of connected components of a space. This is very coarse, as for example it does not distinguish between a disk and a circle, which are both connected.

We can refine this invariant by considering both the connected components, and the associated fundamental group for each of the connected components. More abstractly, this invariant can be obtained by taking the skeleton of the fundamental groupoid. The resulting category is often quite small compared to the fundamental groupoid, while retaining much information about the original groupoid. In the example of the disk, whose fundamental groupoid is infinite, the skeleton only has a single object, and a single morphism. For the circle, the skeleton of the fundamental groupoid is reduced to the fundamental group $\mathbb{Z}$.

For directed topological spaces, the notion of fundamental groupoid is replaced by the fundamental category, and the operation of taking the skeleton has to be replaced by the notion of category of components. This notion was introduced in [20] and is now well understood for loop-free categories [28]. Indeed, transposing the category of components in the directed setting is non-trivial: the fundamental category of a d-space often has no non-trivial isomorphisms, meaning that the category is isomorphic to its skeleton. This means that it is necessary to collapse a wider class of morphisms in order to obtain a more compact representation. Intuitively, we collapse all the morphisms which do not change the future of the past of other morphisms.

Definition 2.2.1. Given a category $\mathcal{C}$, we say that an object y is in the *future* (resp. *past*) of an object x if $\mathcal{C}(x,y) \neq \emptyset$ (resp. $\mathcal{C}(y,x)) \neq \emptyset$.

Given a morphism $f : x \to y$ in a category $\mathcal{C}$, we write $f_* : \mathcal{C}(y,z) \to \mathcal{C}(x,z)$ (resp. $f^* : \mathcal{C}(z,x) \to (z,y)$) for the pre-composition (resp. post-composition) by the morphism f. Our notion of "inessential morphism" will be the following one:

Definition 2.2.2. A morphism $f\colon x \to y$ is a *weak isomorphism* if

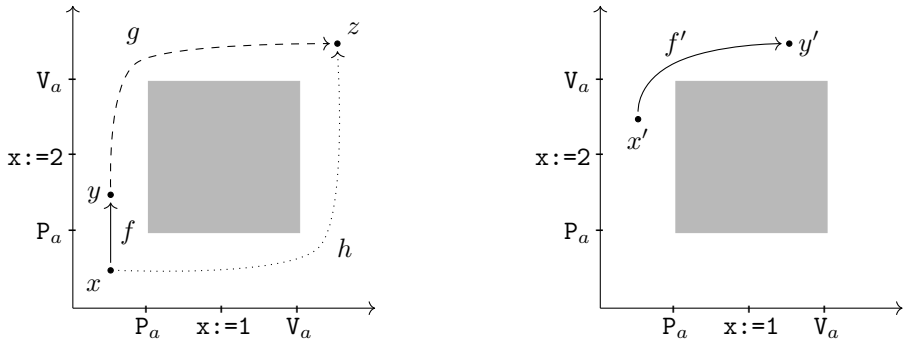
- for any $z \in \mathcal{C}$, such that $\mathcal{C}(y,z) \neq \emptyset$, $f_*\colon \mathcal{C}(y,z) \to \mathcal{C}(x,z)$ is a bijection,
- for any $z \in \mathcal{C}$, such that $\mathcal{C}(z,x) \neq \emptyset$, $f^*\colon \mathcal{C}(z,x) \to \mathcal{C}(z,y)$ is a bijection.

Remark 2.2.3. For any isomorphism f, both f_* and f^* are bijections: any isomorphism is a weak isomorphism.

Example 2.2.4. Let us consider the category $\mathcal{C} = \vec{\Pi}_1(\overline{\mathcal{G}}_P)$ associated to the fundamental category of the directed semantics of the program

$$P = \mathtt{P}_a; \mathtt{x:=1}; \mathtt{V}_a \| \mathtt{P}_a; \mathtt{x:=2}; \mathtt{V}_a$$

Let us focus on the morphisms $f\colon x \to y$ and $f'\colon x' \to y'$ given below.



We can see that f is not a weak isomorphism. Indeed, the morphism $f_*(g) = g \circ f \in \mathcal{C}(x, z)$ is not dihomotopic to the path $h \in \mathcal{C}(x, z)$. Thus, f_* cannot be an isomorphism. It can be shown that the morphism on the right is a weak isomorphism.

Intuitively, the fact that f is not a weak isomorphism corresponds to the fact that when "executing" f, we made an irreversible choice during our execution, namely here `x:=2` will be executed first, changing the value of x in the final state. Then f' being a weak isomorphism could be interpreted from a computing point of view as being the only possible path, up to dihomotopy, between x' and y'.

We have seen that programs with while loops are difficult to handle so that it makes sense to first discard them. A similar phenomenon also occurs for categories, motivating the introduction of the following definition of loop-free category [18, Section 6.2, p. 107].

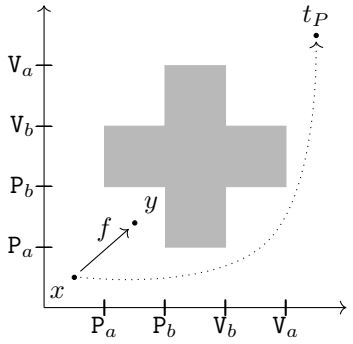
Definition 2.2.5. A morphism $f : x \to y$ in a category $\mathcal{C}$ is said to be *without return* when the hom-set $\mathcal{C}(y, x)$ is empty. Otherwise, we say that f *admits a return*. A category $\mathcal{C}$ is said to be *loop-free* when all its morphisms, except identities, are without return. We write **LFCat** for the category of all small loop-free categories.

Lemma 2.2.6. *Given a loop-free category $\mathcal{C}$, a morphism $f : x \to y$ is a weak isomorphism implies that $\mathcal{C}(x, y) = \{f\}$.*

Proof. See [18, Lemma 6.6]. □

However, this is not exactly what we want from this class of morphisms. Indeed, being the only execution possible from a point to another is not the same as not having made any irreversible choice, i.e. our notion of inessential transitions, as shown in the example Example 2.2.7.

Example 2.2.7. Let us revisit the Swiss Cross Example 1.3.50.



The morphism f drawn above is a weak isomorphism, but should not be considered as inessential. Indeed, executing along f makes an "irreversible choice": the terminal position cannot be reached from y but can be from x.

In order to rule out morphisms that are weak isomorphisms, but do not correspond to inessential transitions, we need to restrict the class of weak isomorphisms we are considering by imposing further restrictions to make it closer to the class of isomorphisms. One property of isomorphisms is that they are stable under pushouts and pullbacks, which is not the case for weak isomorphisms in general. And indeed, the collections of weak

isomorphisms satisfying this property rule out the previous Example 2.2.7 and correspond to our notion of "inessential executions".

Definition 2.2.8. Given a category $\mathcal{C}$, a *system of weak isomorphisms* is a collection Σ of weak isomorphisms of $\mathcal{C}$, which is stable under pushouts and pullbacks, and contains all isomorphisms. We write $\mathbf{SWI}(\mathcal{C})$ the collection of all such systems.

Since weak isomorphisms are stable under composition, we can assume that all systems of weak isomorphisms Σ considered are closed under composition.

Example 2.2.9. In the geometric semantics of the Swiss Cross in Example 2.2.7, the morphism f cannot be in a collection stable by pushout as there is no pushout with any path $g\colon x \to t_P$. For example, we can see that there is no possible pushout between f and the dotted path in Example 2.2.7

The set $\mathbf{SWI}(\mathcal{C})$ is partially ordered by inclusion. This gives rise to a complete lattice structure, where greatest lower bound is the set-theoretic intersection and the greatest upper bound is the smallest system w.r.t. inclusion containing the set-theoretic union.

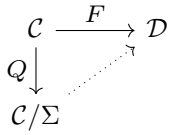
Proposition 2.2.10. *The set* $\mathbf{SWI}(\mathcal{C})$ *forms a complete lattice, with maximal element noted* Σ^m.

Proof. [18, Proposition 6.12] □

For loop-free categories which are finitely presented, such as fundamental categories of loop-free programs, this maximal system can be computed algorithmically by computing the pushouts from the coarsest cubical partition compatible with the normal cover. This is discussed in more details in [18, Section 6.1.4]

This maximal class of weak isomorphisms is precisely what we are aiming to "remove" out of our fundamental category: morphisms that preserve past and future while not interfering with "choices" (i.e. pullback and pushouts). This operation is done by taking the quotient of our category by this maximal class.

Definition 2.2.11. The *quotient* of a category $\mathcal{C}$ by a set of morphisms Σ is a category $\mathcal{C}/\Sigma$ together with a *quotient functor* $Q\colon \mathcal{C} \to \mathcal{C}/\Sigma$ sending all morphisms in Σ to identities and such that any functor $F\colon \mathcal{C} \to \mathcal{D}$ factors uniquely through Q if and only if it sends all morphisms in Σ to identities.



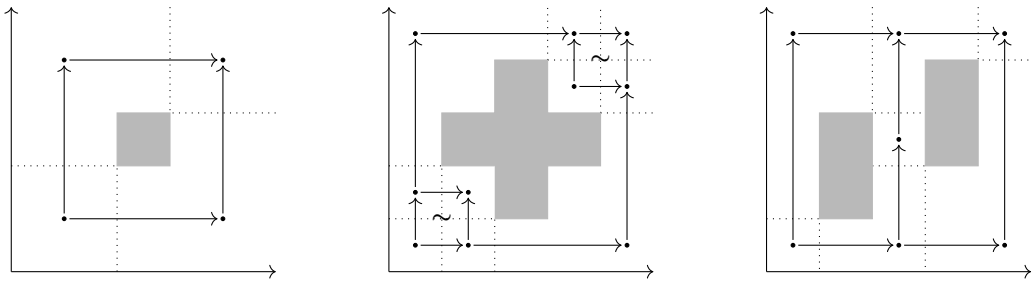
Such a category always exists, see [6], but the resulting category does not necessarily preserve the properties we care about.

Example 2.2.12. Let us consider the two following categories. On the left we have a category $\mathcal{G}$ with two objects E, V and two arrows s, t from E to V. Taking the quotient $\mathcal{G}/\{t\}$ gives us the category the free category over the graph on the right, which has a countable number of arrows.

Definition 2.2.13. The *category of components* $\vec{\Pi}_0(\mathcal{C})$ of a loop-free category $\mathcal{C}$ is the quotient category $\mathcal{C}/\Sigma^m$, where Σ_m is the greatest system of weak isomorphisms of $\mathbf{SWI}(\mathcal{C})$. Given a directed topological space X, we write $\vec{\Pi}_0(X)$ for $\vec{\Pi}_0(\vec{\Pi}_1(X))$.

When Σ is not maximal, the category $\mathcal{C}/\Sigma$ can be thought of an over-approximation of the category of components $\vec{\Pi}_0(\mathcal{C})$, in the sense that the latter is a quotient of the former.

Example 2.2.14. Below are figured three directed spaces corresponding to geometric realizations of programs, together with the generators of their category of components in superimposition.



Remark 2.2.15. Another possible way of defining the category of components w.r.t. the system of weak isomorphisms consists in using the localization [18, Definition 6.25] instead of the quotient. Both these definitions coincide on loop-free categories [18, Theorem 6.30], and none of these definitions give the expected results on categories with loops.

2.2.2 Computing the category of components of loop-free programs

The methods for computing the category of components given in [28] and [18] are both extremely costly, and it makes sense to try to reduce the size of the categories when we can. We have seen in Section 2.1 a method to factor programs in terms of independent processes (Definition 2.1.1), whose associated geometric semantics can then be factored, i.e. for two independent programs P, Q,

$$\overline{\mathcal{G}}_{P\|Q} = \overline{\mathcal{G}}_P \times \overline{\mathcal{G}}_Q$$

Additionally, the functor computing the fundamental category and the one computing the category of components both preserve binary products.

Proposition 2.2.16. *The fundamental category functor* $\vec{\Pi}_1 \colon \mathbf{dTop} \to \mathbf{Cat}$ *preserves binary products i.e.*

$$\vec{\Pi}_1(X \times Y) = \vec{\Pi}_1(X) \times \vec{\Pi}_1(Y)$$

Proof. See [29, Proposition 5.2.12]. □

Proposition 2.2.17. *The category of components functor* $\vec{\Pi}_0 \colon \mathbf{LFCat} \to \mathbf{LFCat}$ *preserves binary products, i.e.*

$$\vec{\Pi}_0(\mathcal{C} \times \mathcal{D}) = \vec{\Pi}_0(\mathcal{C}) \times \vec{\Pi}_0(\mathcal{D})$$

Proof. See [29, Proposition 8.5.11]. □

Thus, the fundamental category $\vec{\Pi}_1(\overline{\mathcal{G}}_{P\|Q})$ and category of components $\vec{\Pi}_0(\overline{\mathcal{G}}_{P\|Q})$ of a loop-free program $P\|Q$, where P and Q are independent, can be computed as the product of their category of components.

Proposition 2.2.18. *Given two independent programs* P, Q *i.e. such that* $\overline{\mathcal{G}}_{P\|Q} = \overline{\mathcal{G}}_P \times \overline{\mathcal{G}}_Q$*, we have:*

$$\vec{\Pi}_1(\overline{\mathcal{G}}_{P\|Q}) = \vec{\Pi}_1(\overline{\mathcal{G}}_P) \times \vec{\Pi}_1(\overline{\mathcal{G}}_Q)$$
$$\vec{\Pi}_0(\overline{\mathcal{G}}_{P\|Q}) = \vec{\Pi}_0(\overline{\mathcal{G}}_P) \times \vec{\Pi}_0(\overline{\mathcal{G}}_Q)$$

Another approach to the factorization of programs would be to directly find factorizations of its model, and from there deducing a factorization of the program. The category of components, being finite, is a good candidate to try and find a product decomposition.

In the case of partial orders, the existence of a refinement for any pair of isomorphic product decomposition is a well-known result to which we refer as Hashimoto's theorem [26]. The category of (small) loop-free categories contains all the partially ordered sets, all the fundamental categories, and all the categories of component of the models of loop-free programs. These remarks motivate our successful attempt to extend Hashimoto's theorem to connected loop-free categories (Section 2.3). Theorem 2.3.36 even guarantees the existence of a maximal refinement for any finite loop-free category (in particular, for any category of component of a loop-free program).

2.3 Factoring loop-free categories

In this section we will introduce Hashimoto's Theorem (Theorem 2.3.3), which states that, any pair of decompositions of a *connected* partial order have a common refinement. This is what we call the *strict refinement property* (Definition 2.3.2). We prove in Theorem 2.3.36 that this property extend to the more general case of loop-free categories. The structure of our proof diverges from Hashimoto's original paper [26] and instead follows more closely the presentation found in [49, Chapter 10].

We say that a poset is connected when any two pair of elements is connected by a "zigzag" of elements:

Definition 2.3.1. A poset is *connected* when for all elements x and y we have a sequence $x = z_0, z_1, \ldots, z_n = y$ such that z_i and z_{i-1} are *comparable* for every $i \in \{1, \ldots, n\}$.

Definition 2.3.2. A connected poset satisfies the *strict refinement property* if when the following isomorphism holds in the category **Pos** of posets:

$$\prod_{\alpha \in A} X_\alpha \quad \cong \quad \prod_{\beta \in B} Y_\beta$$

we have a family of posets $\left\{ Z_{\alpha,\beta} \;\middle|\; \alpha \in A \,;\, \beta \in B \right\}$ such that the isomorphisms

$$X_\alpha \quad \cong \quad \prod_{b \in B} Z_{\alpha,b} \qquad \text{and} \qquad Y_\beta \quad \cong \quad \prod_{a \in A} Z_{a,\beta}$$

hold for every $\alpha \in A$ and every $\beta \in B$.

Theorem 2.3.3 (Hashimoto's Theorem, [26])**.** *Every connected poset has the strict refinement property*

A poset X is said to be *irreducible* when $X \cong A \times B$ implies that either A or B has a single element (but not both!). One easily shows that every finite connected poset can be written as a finite product of *irreducible* posets in unique way, up to reordering of factors. Indeed, for a finite poset, there exists a finite number of product decompositions, which are all isomorphic by definition. With the strict refinement property, we can then construct the refinement of all decomposition in a finite number of steps, which is still a product decomposition and cannot be refined any further.

Example 2.3.4. Let us consider the two following isomorphic decomposition of $\mathbb{R}^3$:

$$\mathbb{R}^2 \times \mathbb{R} = X_1 \times X_2 \qquad\qquad \mathbb{R} \times \mathbb{R}^2 = Y_1 \times Y_2$$

Then $\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R} \times \mathbb{R} \times \{*\} \times \mathbb{R}$ is a refinement of these two decompositions by assigning:

$$\begin{aligned} Z_{1,1} &= \mathbb{R} & \qquad Z_{1,2} &= \mathbb{R} \\ Z_{2,1} &= \{*\} & \qquad Z_{2,2} &= \mathbb{R} \end{aligned}$$

2.3.1 Properties of loop-free categories

Given an object X of a category $\mathcal{C}$, we write id_X for the identity morphism on X. We generally omit the subscript when clear from the context. Given a morphism $f : X \to Y$, we write $\overline{f} = X$ (resp. $\underline{f} = Y$) for its *source* (resp. *target*).

The category of loop-free categories **LFCat** (Definition 2.2.5) is a full subcategory of the category of small categories **Cat**. **LFCat** contains the category **Pos** of partial orders as a full subcategory. The embedding **Pos** $\hookrightarrow$ **LFCat** has a left adjoint which to a category associates the poset obtained by identifying any two arrows with the same source and the same target.

Lemma 2.3.5. *Given two morphisms f, g of a loop-free category $\mathcal{C}$, we have that $f \circ g = \mathrm{id}$ implies $f = g = \mathrm{id}$*

Proof. If $f \circ g = \mathrm{id}$, then both f and g have a return; it follows that both are identities because $\mathcal{C}$ is loop-free. □

Definition 2.3.6. Given a category $\mathcal{C}$ and any set of its objects $\{X_i\}_{\{i \in I\}}$, the *product* of $\{X_i\}_{\{i \in I\}}$ is, if it exists an object denoted

$$\prod_{i \in I} X_i \in \mathcal{C}$$

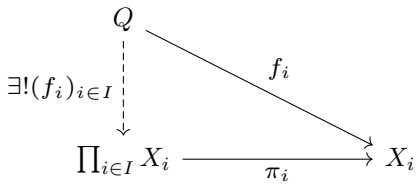
and equipped with morphisms

$$\pi_j \colon \prod_{i \in I} X_i \to X_j$$

called *projections* for all $j \in I$, such that for any family of morphisms $\{f_i \colon Q \to X_i\}_{i \in I}$, there exists a unique morphism

$$(f_i)_{i \in I} \colon Q \to \prod_{i \in I} X_I$$

such that all the following diagrams commutes:



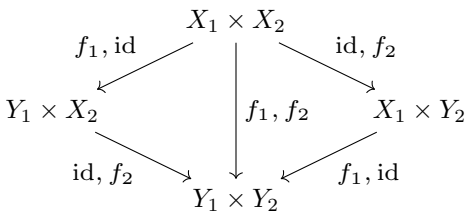
Proposition 2.3.7. LFCat *has all products.*

Proof. **LFCat** is an epireflective subcategory of **Cat** ([28, Proposition 1.8]), a cartesian closed category. Thus, it has all products. □

Definition 2.3.8. In **LFCat**, consider a family $\{f_\alpha : X_\alpha \to Y_\alpha\}_{\alpha \in A}$ of morphisms. The *product map* $f : \prod_{\alpha \in A} X_\alpha \to \prod_{\alpha \in A} Y_\alpha$ is the unique map such that $\pi_\alpha f = f_\alpha$ for every $\alpha \in A$.

For the rest of the chapter we will often write $f = (f_\alpha)_{\alpha \in A}$ when f is the product map of the family $\{f_\alpha : X_\alpha \to Y_\alpha\}_{\alpha \in A}$. For binary products, we write (f, g) the product map of f and g

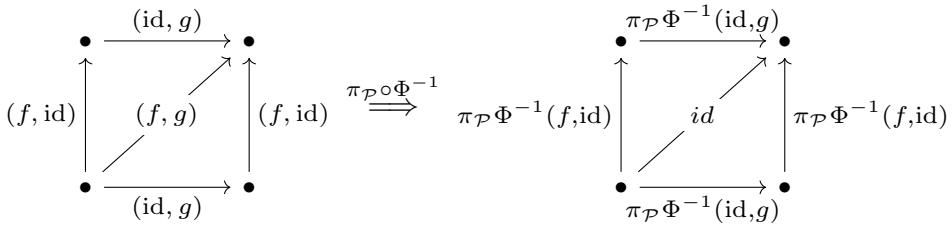
Proposition 2.3.9. *Given a category $\mathcal{C}$ with all products, and two morphisms $f_1 \colon X_1 \to Y_1$ and $f_2 \colon X_2 \to Y_2$ of $\mathcal{C}$, the following diagram commutes:*



Lemma 2.3.10. *Given an isomorphism of small, loop-free categories $\Phi \colon \mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}_1 \times \mathcal{D}_2$ and an object X of $\mathcal{C}_1$. For all morphisms (f, g) of $\mathcal{D}_1 \times \mathcal{D}_2$ such that $\pi_{\mathcal{P}} \Phi^{-1}(f, g) = \mathrm{id}_X$, we have:*

$$\pi_{\mathcal{P}} \Phi^{-1}(f, \mathrm{id}_{\underline{g}}) = \pi_{\mathcal{P}} \Phi^{-1}(f, \mathrm{id}_{\overline{g}}) = \pi_{\mathcal{P}} \Phi^{-1}(\mathrm{id}_{\underline{f}}, g) = \pi_{\mathcal{P}} \Phi^{-1}(\mathrm{id}_{\overline{f}}, g) = \mathrm{id}_X.$$

Proof. By definition of the product, the following diagram commutes:



The category $\mathcal{P}$ is loop-free, thus by applying Lemma 2.3.5 we have

$$\pi_{\mathcal{P}}\Phi^{-1}(\mathrm{id}_{\overline{f}}, g) = \pi_{\mathcal{P}}\Phi^{-1}(f, \mathrm{id}_{\underline{g}}) = \mathrm{id}$$

We prove the same way that $\pi_{\mathcal{P}}\Phi^{-1}(\mathrm{id}_{\underline{f}}, g) = \pi_{\mathcal{P}}\Phi^{-1}(f, \mathrm{id}_{\overline{g}}) = \mathrm{id}$. □

Definition 2.3.11. A *fence* is a family $(f_i)_{1 \leq i \leq n}$ of morphisms of $\mathcal{C}$ such that:

- $\underline{f_{2i}} = \underline{f_{2i+1}}$ and $\overline{f_{2i-1}} = \overline{f_{2i}}$
- or $\overline{f_{2i}} = \overline{f_{2i+1}}$ and $\underline{f_{2i-1}} = \underline{f_{2i}}$

Definition 2.3.12. Two morphisms f and g of a category $\mathcal{C}$ are said to be *connected* if there exists a fence $(f_i)_{i \in [0,n]}$ such that $f = f_0$ and $f_n = g$. A category $\mathcal{C}$ is said to be connected if all its morphisms are connected.

Proposition 2.3.13. *The set of objects* $\mathsf{Obj}(\mathcal{C})$ *of a (small) loop-free category* $\mathcal{C}$ *is equipped with a partial order* $\leq$ *defined by*

$$X \leq Y \iff \mathcal{C}(X, Y) \neq \emptyset.$$

Proof. Let X, Y, Z in $\mathsf{Obj}(\mathcal{C})$.

- Symmetry: $\mathrm{id}_X \in \mathcal{C}(X, X)$, thus $X \leq X$.
- Transitivity: $X \leq Y$ and $Y \leq Z$ implies the existence of $f \in \mathcal{C}(X, Y)$ and $g \in \mathcal{C}(Y, Z)$. Thus, $f \circ g \in \mathcal{C}(X, Z)$ i.e. $X \leq Z$.
- Anti-symmetry: Directly by the loop-freeness property.

□

Proposition 2.3.14. *For* $\mathcal{C}$ *in* **LFCat**, $\mathcal{C}$ *is connected implies* $\mathsf{Obj}(\mathcal{C})$ *is connected as a poset.*

Proof. Let $X, Y \in \mathsf{Obj}(\mathcal{C})$. By definition, $\mathrm{id}_X, \mathrm{id}_Y \in \mathcal{C}$, connected category. Such that there exists a fence $(f_i)_{i \in I}$ between $\mathrm{id}_X, \mathrm{id}_Y$. Then $\overline{f_i} = \overline{f_i + 1}$ implies $\underline{f_i} \leq \overline{f_i} \geq \underline{f_{i+1}}$ by definition of the order. Thus, if we fix $(X_i)_{1 \leq i \leq 2n}$ with $X_{2k} = \overline{f_k}$ and $X_{2k+1} = \underline{f_{k+1}}$, we have a sequence $X_1 \geq X_2 \leq \ldots \geq X_{2n-1} \leq X_{2n}$ connecting $X, Y \in \mathsf{Obj}(\mathcal{C})$. □

Proposition 2.3.15. *Let* $\Phi \colon \mathcal{C} \to \mathcal{D}$ *be a morphism of connected loop-free categories. Let* $\leq$ *be the canonical order defined in Proposition 2.3.13. Then*

$$\Phi|_{\mathsf{Obj}} \colon (\mathsf{Obj}(\mathcal{C}), \leq) \to (\mathsf{Obj}(\mathcal{D}), \leq)$$

is an order-preserving morphism of connected posets. Furthermore, if Φ *is an isomorphism, so is* $\Phi|_{\mathsf{Obj}}$.

Proof. Let X, Y in $\mathsf{Obj}(\mathcal{C})$ such that $X \leq Y$. Thus, there exists $f \in \mathcal{C}(X,Y)$ and by functoriality of Φ, $\Phi(f) \in \mathcal{D}(\Phi(X), \Phi(Y))$ i.e. $\Phi(X) \leq \Phi(Y)$. □

Remark 2.3.16. The property above comes from the adjunction between **LFCat** and **Pos** talked about at the beginning of the section.

2.3.2 Hashimoto's theorem for loop-free categories

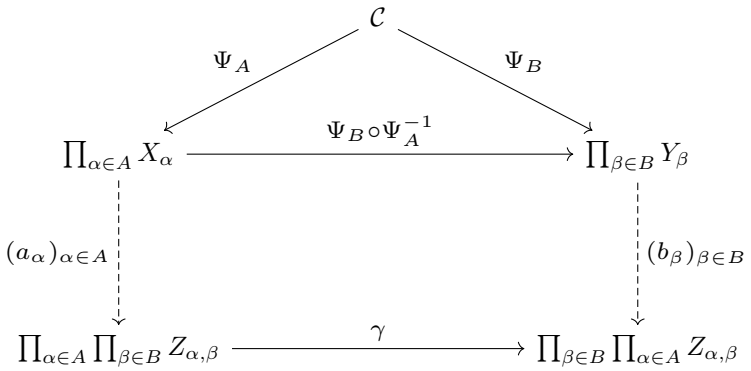
Definition 2.3.17. A *(product) decomposition* of a loop-free category $\mathcal{C}$ is an isomorphism Ψ from $\mathcal{C}$ to a product $\prod_{\alpha\in A} X_\alpha$ of loop-free categories X_α for $\alpha \in A$.

Note that if $\mathcal{C}$ is non-empty, connected, and loop-free, then so are the categories X_α.

Definition 2.3.18. A category $\mathcal{C}$ is said to have the *strict refinement property* if for any two decompositions $\Psi_A : \mathcal{C} \to \prod_{\alpha\in A} X_\alpha$ and $\Psi_B : \mathcal{C} \to \prod_{\beta\in B} Y_\beta$, there exist a family of loop-free categories $Z_{\alpha,\beta}$ with $\alpha \in A$ and $\beta \in B$, and for every $\alpha \in A$ and $\beta \in B$, decompositions

$$a_\alpha \,:\, X_\alpha \to \prod_{\beta\in B} Z_{\alpha,\beta} \quad \text{and} \quad b_\beta \,:\, Y_\beta \to \prod_{\alpha\in A} Z_{\alpha,\beta}$$

such that the following diagram commutes:



(2.1)

Where $\gamma : \prod_{\alpha\in A}\prod_{\beta\in B} Z_{\alpha,\beta} \to \prod_{\beta\in B}\prod_{\alpha\in A} Z_{\alpha,\beta}$ is the natural isomorphism sending $((z_{\alpha,\beta})_{\beta\in B})_{\alpha\in A}$ to $((z_{\alpha,\beta})_{\alpha\in A})_{\beta\in B}$.

Hashimoto's Theorem [26] states that every connected poset has the strict refinement property; we generalize this result to all connected loop-free categories:

Theorem 2.3.36. *Every connected loop-free category has the strict refinement property.*

As previously stated, we will follow the proof from [49], generalizing the key lemmas to the case of connected loop-free categories. We will first give a rough idea of the different steps of the proofs, before introducing some key technical lemmas, that will be used along the proof, before proving all the major steps.

2.3.2.1 Sketch of the proof

To prove Theorem 2.3.36 we will need to find the decompositions $(a_\alpha)_{\alpha\in A}$ and $(b_\beta)_{\beta\in B}$ of Eq. (2.1). For this we will first define some notions, that will be used in the presentation of the proof.

Definition 2.3.19. Given an element s of a set product $\prod_{i\in I} S_i$, an index $j \in I$, and an element $x_j \in S_j$, we define $(s, x_j, j)_{i\in I}$ as the element of $\prod_{i\in I} S_i$ obtained by substituting x_j to s_j in s; in other words:

$$(s, x_j, j)_i = \begin{cases} x_j & \text{if } i = j \\ s_i & \text{otherwise.} \end{cases}$$

Definition 2.3.20. Given an object s of a loop-free category $\prod_{\alpha\in A} X_\alpha$, and an index $\lambda \in A$, the *λ-section at s* is the functor from X_λ to $\prod_{\alpha\in A} X_\alpha$ defined by

$$\begin{cases} x_\lambda & \mapsto & (s, x_\lambda, \lambda) & \text{if } x_\lambda \text{ is an object} \\ f_\lambda & \mapsto & (\mathrm{id}_s, f_\lambda, \lambda) & \text{if } f_\lambda \text{ is a morphism} \end{cases}$$

We denote by $\Xi^s_\lambda : X_\lambda \cong X^s_\lambda$ the isomorphism induced by the λ-section at s on its image. We denote by Φ^s_λ the restriction of Φ to X^s_λ for every functor Φ defined over $\prod_{\alpha\in A} X_\alpha$.

Definition 2.3.21. Given an isomorphism $\Phi\colon \prod_{\alpha\in A} X_\alpha \to \prod_{\beta\in B} Y_\beta$, indexes $\lambda \in A$, $\mu \in B$, and $s \in \prod_{\alpha\in A} X_\alpha$, we define

$$X^\mu_\lambda \coloneqq \pi_\lambda \Phi^{-1} \Xi^{\Phi(s)}_\mu(Y_\mu) \quad \text{and} \quad Y^\lambda_\mu \coloneqq \pi_\mu \Phi \Xi^s_\lambda(X_\lambda)$$

with $\pi_\lambda : \prod_{\alpha\in A} X_\alpha \to X_\lambda$ and $\pi_\mu : \prod_{\beta\in A} X_\beta \to X_\mu$ the projections.

As we will see, the refinement (but not its existence) $Z_{\alpha,\beta}$ (with $\alpha \in A$ and $\beta \in B$) depends on an object $s \in \prod_{\alpha\in A} X_\alpha$ that we arbitrarily fix now. In [49], which proves the strict refinement property for connected posets, the isomorphisms $(a_\alpha)_{\alpha\in A}$ and $(b_\beta)_{\beta\in B}$ of Diagram 2.1 are obtained by decomposing the expected isomorphisms into smaller morphisms, obtained with the notation introduced above, which gives the Diagram 2.2 below.

$$\begin{array}{ccc}
\prod_{\alpha\in A} X_\alpha & \xrightarrow{\quad\Phi\quad} & \prod_{\beta\in B} Y_\beta \\
\downarrow{\scriptstyle (\Xi^s_\alpha)_{\alpha\in A}} & & \downarrow{\scriptstyle (\Xi^{\Phi(s)}_\beta)_{\beta\in B}} \\
\prod_{\alpha\in A} X^s_\alpha & & \prod_{\beta\in B} Y^{\phi(s)}_\beta \\
\downarrow{\scriptstyle (\Phi|_{X^s_\alpha})_{\alpha\in A}} & & \downarrow{\scriptstyle (\Phi^{-1}|_{X^{\Phi(s)}_\beta})_{\beta\in B}} \\
\prod_{\alpha\in A}\prod_{\beta\in B} Y^\alpha_\beta & & \\
\downarrow{\scriptstyle ((\pi_\alpha\Phi^{-1}\Xi^{\Phi(s)}_\beta)_{\beta\in B})_{\alpha\in A}} & & \\
\prod_{\alpha\in A}\prod_{\beta\in B} X^\beta_\alpha & \xleftarrow{\quad\gamma^{-1}\quad} & \prod_{\beta\in B}\prod_{\alpha\in A} X^\beta_\alpha
\end{array} \tag{2.2}$$

The first part of the proof is to prove that all the introduced morphisms, are in fact isomorphisms and to prove that they are indeed product maps of isomorphisms. That is to say that we have $\Psi_2 = \prod\limits_{\alpha\in A} a_\alpha$ and $\Psi_1 = \prod\limits_{\beta\in B} b_\beta$ with

$$a_\alpha = \left(\pi_\alpha\Phi^{-1}\Xi^{\Phi(s)}_\beta\pi_\beta\Phi\Xi^s_\alpha\right)_{\beta\in B} \qquad \text{and} \qquad b_\beta = \Phi^{-1}\Xi^{\Phi(s)}_\beta$$

Most of the proofs rely heavily on the following proposition, which allows to restrain the coordinates of composite images

Lemma 2.3.25. *Let $\Phi\colon \mathcal{P}\times\mathcal{Q}\to\mathcal{U}\times\mathcal{V}$ an isomorphism of connected loop-free categories. Let (u,v) and (u',v') two morphisms of $\mathcal{U}\times\mathcal{V}$ and p a morphism of $\mathcal{P}$. Then $\pi_\mathcal{P}\Phi^{-1}(u,v) = \pi_\mathcal{P}\Phi^{-1}(u',v') = p$ implies $\pi_\mathcal{P}\Phi^{-1}(u,v') = p$.*

Then, once this has been achieved, we need to prove that the diagram commutes to conclude the proof. To do this, we will first prove the commutation on a restriction of the diagram, by replacing for each $\alpha\in A$ the starting category X_α by the subcategory X^μ_α as in Definition 2.3.21, for an arbitrary $\mu\in\beta$, giving us the diagram below.

$$\begin{array}{ccc}
\prod_{\alpha\in A} X^\mu_\alpha & \xrightarrow{\quad\Phi\quad} & Y^{\phi(s)}_\mu \\
\downarrow{\scriptstyle \Psi_2} & & \downarrow{\scriptstyle \Psi_1} \\
\prod_{\alpha\in A}\prod_{\beta\in B} X^\beta_\alpha & \xleftarrow{\quad\gamma^{-1}\quad} & \prod_{\beta\in B}\prod_{\alpha\in A} X^\beta_\alpha
\end{array} \tag{2.3}$$

Then using the following Lemma 2.3.24, the commutation is extended along a given $\lambda\in A$.

Lemma 2.3.24. *Let $\Phi\colon \mathcal{P}\times\mathcal{Q}\to\mathcal{U}\times\mathcal{V}$. Let $\mathcal{Q}'$ a connected subcategory of $\mathcal{Q}$. Let p a morphism of $\mathcal{P}$. Then $\pi_U\Phi(p,q)=\pi_U\Phi(p,q')$ for all $q,q'\in\mathcal{Q}'$ implies, for all $p'\in\mathcal{P}$, $\pi_U\Phi(p',q)=\pi_U\Phi(p',q')$ for all $q,q'\in\mathcal{Q}'$*

This leads to the commutation of the Diagram 2.4 below.

$$\begin{array}{ccc} \prod_{\alpha\in A\setminus\lambda} X_\alpha^\mu \times X_\lambda & \xrightarrow{\quad\Phi\quad} & \prod_{\beta\in B} Y_\beta \\ \Big\downarrow{\scriptstyle (\pi_\alpha\Psi_2)_{\alpha\neq\lambda}\times\pi_\lambda\Psi_2} & & \Big\downarrow{\scriptstyle \Psi_1} \\ \prod_{\alpha\in A}\prod_{\beta\in B} X_\alpha^\beta & \xleftarrow{\quad\gamma^{-1}\quad} & \prod_{\beta\in B}\prod_{\alpha\in A} X_\alpha^\beta \end{array} \tag{2.4}$$

Using Proposition 2.3.23 once more the commutativity can be extended to the full domain $\prod_{\alpha\in A} X_\alpha$, thus ending the proof.

In our case a few necessary conditions that are less trivial will need to be detailed in Lemma 2.3.34 to perform the last steps of the proof, but the broad strokes will remain the same.

2.3.2.2 Technical Lemmas

The proof of Theorem 2.3.3 in [49] makes extensive use of the Proposition 2.3.22 and Proposition 2.3.23, which we will respectively extend to connected elements of **LFCat** in Lemma 2.3.25 and Lemma 2.3.24. As we have changed the formulation to make the proofs easier to follow, we give them here and refer to the original work for the proof.

These lemmas are at the core of the proof and are where the hypothesis that we are using loop-free and connected categories really comes into play, so it is important to keep them in mind.

Proposition 2.3.22. *Given $\Phi\colon P\times Q\to U\times V$ an isomorphism of connected posets and $p\in P$,*

$$\pi_{\mathcal{P}}\Phi^{-1}(u,v)=\pi_{\mathcal{P}}\Phi^{-1}(u',v')=p \text{ implies } \pi_{\mathcal{P}}\Phi^{-1}(u,v')=p$$

Proof. [49, Lemma 10.4.5]. □

Proposition 2.3.23. *Given $\Phi\colon P\times Q\to U\times V$ an isomorphism of connected posets, if there exists $p\in P$ such that $\pi_U\Phi(p,q)=\pi_U\Phi(p,q')$, then for each $p'\in P$,*

$$\pi_U\Phi(p',q)=\pi_U\Phi(p',q')$$

Proof. [49, Lemma 10.4.8] □

In the two following proofs, as the objects in the commutative diagrams are of no importance, we have omitted them, replacing them by • when not necessary.

Lemma 2.3.24. *Let $\Phi\colon \mathcal{P}\times\mathcal{Q}\to\mathcal{U}\times\mathcal{V}$. Let $\mathcal{Q}'$ a connected subcategory of $\mathcal{Q}$. Let p a morphism of $\mathcal{P}$. Then $\pi_U\Phi(p,q)=\pi_U\Phi(p,q')$ for all $q,q'\in\mathcal{Q}'$ implies, for all $p'\in\mathcal{P}$, $\pi_U\Phi(p',q)=\pi_U\Phi(p',q')$ for all $q,q'\in\mathcal{Q}'$*

Proof. Let p a morphism of $\mathcal{P}$ Let $\mathcal{Q}'$ a connected sub-category of $\mathcal{Q}$, such that for all morphisms q, q' in $\mathcal{Q}'$, $\pi_U\Phi(p,q) = \pi_U\Phi(p,q')$. Let us prove that for all p' morphism of $\mathcal{P}$, $\pi_U\Phi(p',q') = \pi_U\Phi(p',q)$ by induction on the length n of the fence connecting q, q'.

- $n = 0$. Trivial.
- $n = 1$. $\pi_U\Phi(p,q) = \pi_U\Phi(p,q')$ for all morphisms q, q' in $\mathcal{Q}'$ implies

$$\begin{aligned}
\underline{\pi_U\Phi(p,q)} &= \underline{\pi_U\Phi(p,q')} \\
\pi_U\Phi(\underline{p},\underline{q}) &= \pi_U\Phi(\underline{p},\underline{q'}) && \text{for all } q, q' \in \mathsf{Mor}(\mathcal{Q}) \\
\pi_U\Phi(\underline{p},Q) &= \pi_U\Phi(\underline{p},Q') && \text{for all } Q, Q' \in \mathsf{Obj}(\mathcal{Q})
\end{aligned}$$

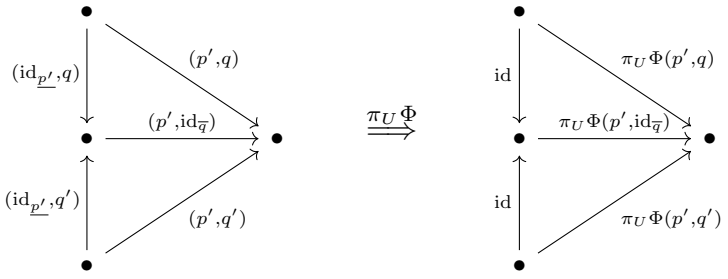
  By Proposition 2.3.15 $\Phi|_{\mathsf{Obj}}$ is an isomorphism between $\mathsf{Obj}(\mathcal{P})\times\mathsf{Obj}(\mathcal{Q})$ and $\mathsf{Obj}(\mathcal{U})\times\mathsf{Obj}(\mathcal{V})$. By Proposition 2.3.23

$$\text{for all } Q, Q' \in \mathsf{Obj}(\mathcal{Q}), \pi_U\Phi(\underline{p'},Q) = \pi_U\Phi(\underline{p'},Q') \tag{2.5}$$

  Thus, $q, q' \in \mathsf{Mor}(\mathcal{Q}')$ implies

$$\begin{aligned}
\underline{\pi_U\Phi(\mathrm{id}_{\underline{p'}},q)} &= \pi_U\Phi(\underline{p'},\underline{q}) \\
&= \pi_U\Phi(\underline{p'},\overline{q}) && \text{By Eq. (2.5)} \\
&= \pi_U\Phi(\overline{\mathrm{id}_{\underline{p'}}},\overline{q}) \\
\underline{\pi_U\Phi(\mathrm{id}_{\underline{p'}},q)} &= \overline{\pi_U\Phi(\mathrm{id}_{\underline{p'}},q)}
\end{aligned}$$

  Let us suppose that $\overline{q} = \overline{q'}$, the other case being dual. By the above argument, the following diagrams commute



  Therefore, $\pi_U\Phi(p',q') = \pi_U\Phi(p',\mathrm{id}_{\overline{q}}) = \pi_U\Phi(p',q)$.

- Now suppose a fence $(q = q_0, q_1 \cdots, q_n = q')$ and the property true for all integers strictly smaller than n. Then there is a $n-1$-fence $(q_1, \cdots, q_n)$ and a 1-fence (q_0, q_1). By induction hypothesis, $\pi_U\Phi(p',q_n) = \pi_U\Phi(p',q_1) = \pi_U\Phi(p',q_0)$.

$\mathcal{Q}'$ is connected so for all $p' \in \mathcal{P}$, for all $q, q' \in \mathcal{Q}'$, $\pi_U\Phi(p',q') = \pi_U\Phi(p',q')$. □

Lemma 2.3.25. *Let $\Phi\colon \mathcal{P}\times\mathcal{Q} \to \mathcal{U}\times\mathcal{V}$ an isomorphism of connected loop-free categories. Let (u,v) and (u',v') two morphisms of $\mathcal{U}\times\mathcal{V}$ and p a morphism of $\mathcal{P}$. Then $\pi_{\mathcal{P}}\Phi^{-1}(u,v) = \pi_{\mathcal{P}}\Phi^{-1}(u',v') = p$ implies $\pi_{\mathcal{P}}\Phi^{-1}(u,v') = p$.*

Proof. Given p, q, q' such that $\Phi(p, q) = (u, v)$ and $\Phi(p, q') = (u', v')$. Let us prove that $\pi_{\mathcal{P}}\Phi^{-1}(u, v') = p$. We proceed by induction on the length of the fence between q and q'.

- A fence of length 2 implies $\underline{q} = \underline{q'}$ or $\overline{q} = \overline{q'}$. First, let us suppose $\underline{q} = \underline{q'}$, the other case being solved dually.
  Let us prove that $\pi_{\mathcal{P}}\Phi^{-1}(u, v') = p$.

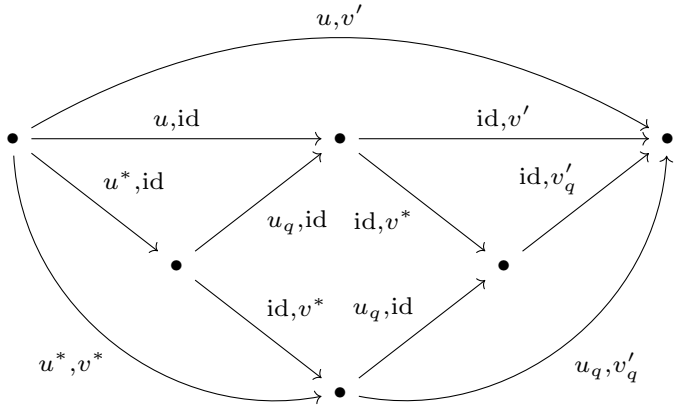
  Define

  $$\Phi(p, \mathrm{id}_{\underline{q}}) := (u^*, v^*) \qquad \Phi(\mathrm{id}_{\overline{p}}, q) := (u_q, v_q) \qquad \Phi(\mathrm{id}_{\overline{p}}, q') := (u'_q, v'_q) \tag{2.6}$$

  such that:

  $$u = u_q \circ u^* \qquad u' = u'_q \circ u^*$$
  $$v = v_q \circ v^* \qquad v' = v'_q \circ v^*$$

  By the Proposition 2.3.9, the following diagram commutes:



  Following the outer arrows, we get

  $$(u, v') = (u_q, v'_q) \circ (u^*, v^*)$$

  By Definition Eq. (2.6), $\pi_{\mathcal{P}}\Phi^{-1}(u^*, v^*) = p$. We are thus left to prove $\pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q) = \mathrm{id}$. By construction $\underline{v_q} = \underline{v'_q} = \overline{v^*}$ such that by functoriality:

  $$\begin{aligned} \underline{\pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q)} &= \pi_{\mathcal{P}}\Phi^{-1}(\underline{u_q}, \underline{v'_q}) \\ &= \pi_{\mathcal{P}}\Phi^{-1}(\underline{u_q}, \underline{v_q}) \\ &= \underline{\pi_{\mathcal{P}}\Phi^{-1}(u_q, v_q)} \\ \underline{\pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q)} &= \overline{p} \end{aligned}$$

By Proposition 2.3.15, $\Phi\colon \mathsf{Obj}(\mathcal{P})\times\mathsf{Obj}(\mathcal{Q}) \to \mathsf{Obj}(\mathcal{U})\times\mathsf{Obj}(\mathcal{V})$ is an order-preserving isomorphism of connected posets and such that $\pi_{\mathcal{P}}\Phi^{-1}(\overline{u_q}, \overline{v_q}) = \pi_{\mathcal{P}}\Phi^{-1}(\overline{u'_q}, \overline{v'_q}) = \overline{p}$. Hence,

$$\begin{array}{ll} \overline{p} = \pi_{\mathcal{P}}\Phi^{-1}(\overline{u_q}, \overline{v'_q}) & \text{Proposition 2.3.22} \\ \overline{p} = \overline{\pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q)} & \text{by functoriality} \end{array}$$

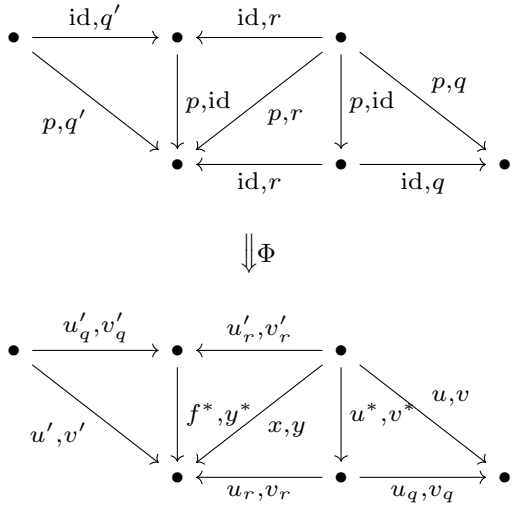
Thus, $\pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q) \in P(\overline{p}, \overline{p})$, with P loop-free. This implies $\pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q) = \mathrm{id}_{\overline{p}}$, such that

$$\begin{aligned} \pi_{\mathcal{P}}\Phi^{-1}(u, v') &= \pi_{\mathcal{P}}\Phi^{-1}(u_q, v'_q) \circ \pi_{\mathcal{P}}\Phi^{-1}(u^*, v^*) \\ \pi_{\mathcal{P}}\Phi^{-1}(u, v') &= p \end{aligned}$$

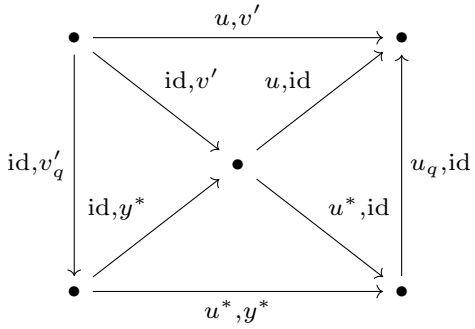
- Now let us suppose a fence $\cdot \xrightarrow{q'} \cdot \xleftarrow{r} \cdot \xrightarrow{q} \cdot$ of length $n = 3$ between q and q'. Let

$$(u, v) := \Phi(p, q) \qquad (x, y) := \Phi(p, r) \qquad (u', v') := \Phi(p, q') \tag{2.7}$$

such that Φ sends the commutative diagram on top to the one below.



We are going to use the same method as before, working on each 2-fence inside the 3-fence above. For that we'll decompose (u, v') using the Diagram Section 2.3.2.2. By the Proposition 2.3.9, the following diagram commutes:



such that

$$\begin{aligned}
(u,v') &= (u_q, \mathrm{id}_{\overline{v'}}) \circ (u^*, y^*) \circ (\mathrm{id}_{\underline{u^*}}, v'_q) && \text{By following the outer arrows}\\
(u,v') &= (u_q, \mathrm{id}_{\overline{v_r}}) \circ (u^*, y^*) \circ (\mathrm{id}_{\underline{u'_r}}, v'_q) && \text{By the Diagram 2.3.2.2}
\end{aligned}$$

– Working on the 2-fence $\overset{q}{\leftarrow} \cdot \overset{r}{\rightarrow}$, let us prove $\pi_{\mathcal{P}}\Phi^{-1}(u_q, \mathrm{id}_{\overline{v_r}}) = \mathrm{id}$
By Definition 2.7 we have $\pi_{\mathcal{P}}\Phi^{-1}(u,v) = \pi_{\mathcal{P}}\Phi^{-1}(x,y) = p$. Thus, as proved for 2-fences above

$$\begin{aligned}
p &= \pi_{\mathcal{P}}\Phi^{-1}(u,y)\\
&= \pi_{\mathcal{P}}\Phi^{-1}((u_q, v_r) \circ (u^*, v^*)) && \text{Diagram 2.3.2.2}\\
&= \pi_{\mathcal{P}}\Phi^{-1}(u_q, v_r) \circ \pi_{\mathcal{P}}\Phi^{-1}(u^*, v^*) && \text{Functoriality}\\
p &= \pi_{\mathcal{P}}\Phi^{-1}(u_q, v_r) \circ p && \text{Diagram 2.3.2.2}
\end{aligned}$$

By loop-free property of $\mathcal{P}$, $\pi_{\mathcal{P}}\Phi^{-1}(u_q, v_r) = \mathrm{id}_{\overline{p}}$. Which implies by Lemma 2.3.10,

$$\pi_{\mathcal{P}}\Phi^{-1}(u_q, \mathrm{id}_{\overline{v_r}}) = \mathrm{id}_{\overline{p}}$$

Similarly, $\pi_{\mathcal{P}}\Phi^{-1}(\mathrm{id}_{\underline{u'_r}}, v'_q) = \mathrm{id}_{\underline{p}}$. Hence

$$\begin{aligned}
\pi_{\mathcal{P}}\Phi^{-1}(u,v') &= \pi_{\mathcal{P}}\Phi^{-1}(u_q, \mathrm{id}_{\overline{v_r}}) \circ \pi_{\mathcal{P}}\Phi^{-1}(u^*, y^*) \circ \pi_{\mathcal{P}}\Phi^{-1}(\mathrm{id}_{\underline{u'_r}}, v'_q)\\
&= \mathrm{id}_{\overline{p}} \circ \pi_{\mathcal{P}}\Phi^{-1}(u^*, y^*) \circ \mathrm{id}_{\underline{p}}\\
\pi_{\mathcal{P}}\Phi^{-1}(u,v') &= \pi_{\mathcal{P}}\Phi^{-1}(u^*, y^*)
\end{aligned}$$

– Let us prove now $\pi_{\mathcal{P}}\Phi^{-1}(u^*, y^*) = p$. By commutativity of both projections the central square:

$$\begin{aligned}
(x,y) &= (u_r \circ u^*, y^* \circ v'_r)\\
&= (u_r, \mathrm{id}_{\overline{u^*}}) \circ (u^*, y^*) \circ (\mathrm{id}_{\underline{v^*}}, v'_r)\\
(x,y) &= (u_r, \mathrm{id}_{\underline{v_r}}) \circ (u^*, y^*) \circ (\mathrm{id}_{\underline{u'_r}}, v'_r)
\end{aligned}$$

$\pi_{\mathcal{P}}\Phi^{-1}(u_r, v_r) = \mathrm{id}_{\overline{p}}$. , by Lemma 2.3.10, $\pi_{\mathcal{P}}\Phi^{-1}(u_r, \mathrm{id}_{\overline{v_r}}) = \mathrm{id}_{\overline{p}}$. Similarly, we have $\pi_{\mathcal{P}}\Phi^{-1}(\mathrm{id}_{\underline{u'_r}}, v'_r) = \mathrm{id}_{\underline{p}}$. Thus

$$p = \pi_{\mathcal{P}}\Phi^{-1}(x,y) = \pi_{\mathcal{P}}\Phi^{-1}(u^*, y^*)$$

Such that

$$\pi_{\mathcal{P}}\Phi^{-1}(u, v') = p$$

- Now let us suppose the property holds for all $k \leq n$, that is for all $p \in P$, and for each fence $(q_i)_{0\leq i\leq n}$, with $(u_i, v_i) \coloneqq \Phi(p, q_i)$,

$$\pi_{\mathcal{P}}\Phi^{-1}(u_0, v_0) = \pi_{\mathcal{P}}\Phi^{-1}(u_k, v_k) = p \text{ implies } \pi_{\mathcal{P}}\Phi^{-1}(u_0, v_k) = p$$

Given $q_0 = q$ and $q_n = q'$ the extremities of a n-fence $(q_i)_{0\leq i\leq n}$. By definition $((p, q_i))_{0\leq i\leq n-1}$ and $((p, q_i))_{1\leq i\leq n}$ are $n-1$-fence.

$$p = \pi_{\mathcal{P}}\Phi^{-1}(u_0, v_0) = \pi_{\mathcal{P}}\Phi^{-1}(u_{n-1}, v_{n-1})$$
$$p = \pi_{\mathcal{P}}\Phi^{-1}(u_1, v_1) = \pi_{\mathcal{P}}\Phi^{-1}(u_n, v_n)$$

Thus by induction hypothesis, this implies

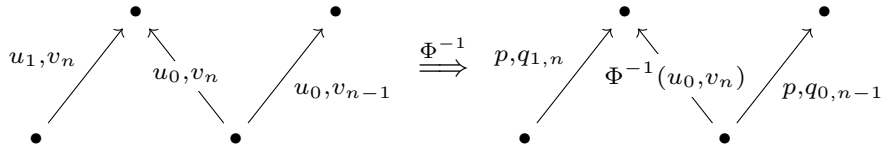
$$p = \pi_{\mathcal{P}}\Phi^{-1}(u_0, v_{n-1}) \qquad\qquad p = \pi_{\mathcal{P}}\Phi^{-1}(u_1, v_n)$$

Now we suppose $\overline{q_0} = \overline{q_1}$ ($\underline{q_0} = \underline{q_1}$ can be treated dually). We'll have to proceed differently depending on the symmetry of the fence.

– If $\overline{q_{n-1}} = \overline{q_n}$, by functoriality of Φ, $\overline{u_{n-1}}, \overline{v_{n-1}} = \overline{u_n}, \overline{v_n}$, such that $\overline{u_1, v_n} = \overline{u_0, v_{n-1}}$.

$$\begin{aligned} p &= \pi_{\mathcal{P}}\Phi^{-1}(u_1, v_n) = \pi_{\mathcal{P}}\Phi^{-1}(u_0, v_{n-1}) \\ &= \pi_{\mathcal{P}}\Phi^{-1}(u_0, v_n) && \text{Induction hypothesis on a 2-fence} \\ p &= \pi_{\mathcal{P}}\Phi^{-1}(u, v') \end{aligned}$$

– If $\underline{q_{n-1}} = \underline{q_n}$, by functoriality of Φ, $\underline{u_{n-1}}, \underline{v_{n-1}} = \underline{u_n}, \underline{v_n}$, and we get the following 2-fence:



By applying the case $n = 2$ of our induction to $\Phi(p, q_{0,n-1}) = (u_0, v_{n-1})$, $\Phi(p, q_{1,n}) = (u_1, v_n)$ we get $\pi_{\mathcal{P}}\Phi^{-1}(u, v') = \pi_{\mathcal{P}}\Phi^{-1}(u_0, v_n) = p$.

Thus, $\pi_{\mathcal{P}}\Phi^{-1}(u, v) = \pi_{\mathcal{P}}\Phi^{-1}(u', v') = p$ implies $\pi_{\mathcal{P}}\Phi^{-1}(u, v') = p$, proving the induction step.

□

This Lemma 2.3.25 is not restricted to binary product and easily extends to arbitrary products as shown in the following lemma.

Corollary 2.3.26. *Let* $\Phi\colon \prod_{\alpha\in A} X_\alpha \to \prod_{\beta\in B} Y_\beta$ *an isomorphism of connected categories. Let* $f^i = (f^i_\alpha)_{\alpha\in A} \in \prod_{\alpha\in A} X_\alpha$ *for* $i = 1, 2$. *Then for all* $\lambda \in A$ *and all* $\beta \in B$ *such that* $\pi_\lambda f^1 = \pi_\lambda f^2$

$$\pi_\lambda \Phi^{-1}(\Phi(f^1), \Phi(f^2)_\beta, \beta) = \pi_\lambda f^1 = \pi_\lambda f^2$$

Proof. This follows directly from Lemma 2.3.25 by taking:

$$P \coloneqq X_\lambda \qquad Q \coloneqq \prod_{\alpha\in A\setminus\{\lambda\}} X_\alpha \qquad U \coloneqq \prod_{\beta\in\{\mu\in B | y_\mu = y^1_\mu\}} Y_\beta \qquad V \coloneqq \prod_{\beta\in\{\mu\in B | y_\mu \neq y^1_\mu\}} Y_\beta$$

$$(p, q) \coloneqq \gamma^{-1}(f^1) \qquad (p, q') \coloneqq \gamma^{-1}(f^2) \qquad (u, v') \coloneqq \delta(g)$$

With γ and δ the natural isomorphisms:

$$\gamma\colon X_\lambda \times \prod_{\alpha\in A\setminus\{\lambda\}} X_\alpha \to \prod_{\alpha\in A} X_\alpha \qquad \delta\colon \prod_{\beta\in B} Y_\beta \to \prod_{\beta\in\{\mu\in B | g_\mu = g^1_\mu\}} Y_\beta \times \prod_{\beta\in\{\mu\in B | g_\mu \neq g^1_\mu\}} Y_\beta$$

We get an isomorphisms $\Psi = \delta \circ \Phi \circ \gamma\colon P \times Q \to U \times V$. Such that $f_\lambda = \pi_{\mathcal{P}} \Psi^{-1}(u, v') = \pi_\lambda \Phi^{-1}(g)$ □

2.3.2.3 Proof of Theorem 2.3.36

For the reminder of section, we will consider that Φ is an isomorphism of connected loop-free categories, $(X_\alpha)_{\alpha\in A}$ and $(Y_\beta)_{\beta\in B}$ families of connected loop-free categories.

As stated before, we will first prove that all the morphisms of the Diagram 2.2 are isomorphisms.

Lemma 2.3.27. *Given a category* $X = \prod_{\alpha\in A} X_\alpha$, $\lambda \in A$, *s a morphism of* X *and* Ξ^s_λ *as defined in Definition 2.3.20, then*

$$\pi_\lambda \circ \Xi^s_\lambda = \mathrm{id}_{X_\lambda}$$

We say that Ξ^s_λ *is a* section *of the canonical projection* $\pi_\lambda\colon \prod_{\alpha\in A} X_\alpha \to X_\lambda$. *Furthermore,* Ξ^s_λ *is a full and faithful functor*

Proof. Let $f \in \prod_{\alpha\in A} X_\alpha((s, x_\lambda, \lambda), (s, y_\lambda, \lambda))$. Then X_α loop-free implies, $\pi_\alpha f = \mathrm{id}_{s_\alpha}$ if $\alpha \neq \lambda$ and $\pi_\lambda f \coloneqq f_\lambda \in X_\lambda(x_\lambda, y_\lambda)$. Such that $f = \Xi^s_\lambda(f_\lambda)$. This proves that Ξ^s_λ is full. Furthermore, it is clearly faithful. □

Corollary 2.3.28. *Given a category* $X = \prod_{\alpha\in A} X_\alpha$, $\lambda \in A$, *s a morphism of* X *then* $X^s_\lambda = \Xi^s_\lambda(X)$ *is a connected, loop-free full sub-category of* X *isomorphic to* X_λ. *Furthermore*

$$\pi_\lambda \circ \Xi^s_\lambda = \mathrm{id}_{X_\lambda} \quad \Xi^s_\lambda \circ \pi_\lambda|_{X^s_\lambda} = \mathrm{id}_{X^s_\lambda}$$

One last proposition that we will need from [49] is the fact that for any $\lambda \in A$, s_λ object of X_λ, the object part of the functor $\Phi|_{X^s_\lambda}\colon X^s_\lambda \to \prod_{\beta\in B} Y^\lambda_\beta$ is an isomorphism of the underlying objects of the category.

Proposition 2.3.29. *[49, Lemma 10.4.7]*
Let $\Phi\colon \prod_{\alpha\in A} X_\alpha \to \prod_{\beta\in B} Y_\beta$ *be an isomorphism of connected posets. Let* $s \in \prod_{\alpha\in A} X_\alpha$ *and* $\lambda \in A$ *and* X^s_λ *be as in Lemma 2.3.27. Let* $Y^\lambda_\beta = \pi_\beta\Phi[X^s_\lambda]$*, then*

$$\Phi|_{X^s_\lambda}\colon X^s_\lambda \to \prod_{\beta\in B} Y^\lambda_\beta$$

is an isomorphism of posets.

This proposition also translates to an isomorphism of connected categories.

Proposition 2.3.30. *Let* $\Phi\colon \prod_{\alpha\in A} X_\alpha \to \prod_{\beta\in B} Y_\beta$ *be an isomorphism of connected loop-free categories. Let* $s \in \prod_{\alpha\in A} X_\alpha$ *and* $\lambda \in A$ *and* X^s_λ *be as in Lemma 2.3.27. Let* $Y^\lambda_\beta = \pi_\beta\Phi[X^s_\lambda]$*, then*

$$\Phi|_{X^s_\lambda}\colon X^s_\lambda \to \prod_{\beta\in B} Y^\lambda_\beta$$

is an isomorphism of connected loop-free categories.

Proof. Let us show that $\Phi|_{X^s_\lambda}\colon X^s_\lambda \to \prod_{\beta\in B} Y^\lambda_\beta$ is essentially surjective. By Proposition 2.3.15, $\Phi_{\mathsf{Obj}}|_{X^s_\lambda}$ is an isomorphism of posets. Thus, by Proposition 2.3.29,

$$\begin{aligned}\Phi_{\mathsf{Obj}}[X^s_\lambda] &= \prod_{\beta\in B} \pi_\beta\Phi_{\mathsf{Obj}}[\mathsf{Obj}(X^s_\lambda)]\\ &= \prod_{\beta\in B} \mathsf{Obj}(\pi_\beta\Phi[X^s_\lambda])\\ \Phi_{\mathsf{Obj}}[X^s_\lambda] &= \mathsf{Obj}(\prod_{\beta\in B} \pi_\beta\Phi[X^s_\lambda])\end{aligned}$$

Thus $\Phi|_{X^s_\lambda}$ is essentially surjective. Furthermore, $\Phi|_{X^s_\lambda}$ is full and faithful as the restriction of a full and faithful functor to a full subcategory (Lemma 2.3.27 and Corollary 2.3.28). Thus, $\Phi|_{X^s_\lambda}$ is a fully faithful and essentially surjective functor, thus an isomorphism in **LFCat** □

Proposition 2.3.31. *Let* $\Phi\colon \prod_{\alpha\in A} X_\alpha \to \prod_{\beta\in B} Y_\beta$*. Fix* $\alpha \in A$ *and* $\beta \in B$*. With*

$$Y^\alpha_\beta \coloneqq \pi_\beta\Phi[X^s_\alpha] \qquad\qquad X^\beta_\alpha \coloneqq \pi_\alpha\Phi^{-1}[Y^{\Phi(s)}_\beta]$$

The two following morphisms are inverse of each other

$$\pi_\alpha \circ \Phi^{-1} \circ \Xi^{\Phi(s)}_\beta\colon Y^\alpha_\beta \to X^\beta_\alpha \qquad\qquad \pi_\beta \circ \Phi \circ \Xi^s_\alpha\colon X^\beta_\alpha \to Y^\alpha_\beta$$

Proof. Let us prove that

$$\pi_\beta\Phi \circ \Xi^s_\alpha\pi_\alpha \circ \Phi^{-1}\Xi^{\Phi(s)}_\beta|_{Y^\alpha_\beta}\colon Y^\alpha_\beta \to Y^\alpha_\beta \text{ is the identity.}$$

$\pi_\beta\Phi \circ \Phi^{-1}\Xi^{\Phi(s)}_\beta = \mathrm{id}_{Y^\alpha_\beta}$ and $\Xi^s_\alpha\pi_\alpha|_{X^s_\alpha} = \mathrm{id}_{X^s_\alpha}$ (Corollary 2.3.28). But this contraction can only be made if we prove that $\Phi^{-1}\Xi^{\Phi(s)}_\beta$ sends Y^α_β to a subcategory of X^s_α. As

$\pi_\beta \Phi \Xi^s_\alpha \colon X_\alpha \to Y^\alpha_\beta$ is full and essentially surjective by Proposition 2.3.30, it is equivalent to proving that $\Phi^{-1}\Xi^{\Phi(s)}_\beta \pi_\beta \Phi \Xi^s_\alpha$ sends X_α to a subcategory of X^s_α

Let $f_\alpha \in X_\alpha$ and $\lambda \in A, \lambda \neq \alpha$. By definition, $\pi_\lambda \mathrm{id}_s = \pi_\lambda \Xi^s_\alpha f$. Thus, by Corollary 2.3.26,

$$\pi_\lambda \Phi^{-1}(\Phi(\mathrm{id}_s), \Phi(\Xi^s_\alpha f)_\beta, \beta) = \pi_\lambda \mathrm{id}_s$$

i.e.
$$\pi_\lambda \Phi^{-1} \Xi^{\Phi(s)}_\beta \pi_\beta \Phi \Xi^s_\alpha f = \pi_\lambda \mathrm{id}_s$$

Thus, $\Phi^{-1}\Xi^{\Phi(s)}_\beta \pi_\beta \Phi \Xi^s_\alpha$ sends X_α to a subcategory of X^s_α. Such that,

$$\begin{aligned}
\pi_\beta \Phi \circ \Xi^s_\alpha \pi_\alpha \circ \Phi^{-1}\Xi^{\Phi(s)}_\beta|_{Y^\alpha_\beta} &= \pi_\beta \Phi \circ \underbrace{\Xi^s_\alpha \pi_\alpha}_{=\mathrm{id}_{X^s_\alpha}} \circ \underbrace{\Phi^{-1}\Xi^{\Phi(s)}_\beta|_{Y^\alpha_\beta}}_{\text{maps to } X^s_\alpha} \\
&= \pi_\beta \circ \Phi \circ \Phi^{-1} \circ \Xi^{\Phi(s)}_\beta|_{Y^\alpha_\beta} \\
&= \pi_\beta \Xi^{\Phi(s)}_\beta|_{Y^\alpha_\beta} \\
\pi_\beta \Phi \circ \Xi^s_\alpha \pi_\alpha \circ \Phi^{-1}\Xi^{\Phi(s)}_\beta|_{Y^\alpha_\beta} &= \mathrm{id}_{Y^\alpha_\beta} \qquad \text{Corollary 2.3.28}
\end{aligned}$$

Similarly, $\pi_\alpha \Phi^{-1}\Xi^{\Phi(s)}_\beta \circ \pi_\beta \Phi \Xi^s_\alpha|_{X^\beta_\alpha} = \mathrm{id}_{X^\beta_\alpha}$ □

With this we have all we need to build the isomorphism $\Psi_1 = (b_\beta)_{\beta\in B} \colon \prod_{\beta\in B} Y_\beta \to \prod_{\alpha\in A,\beta\in B} X^\beta_\alpha$ and $\Psi_2 = (a_\alpha)_{\alpha\in A} \colon \prod_{\alpha\in A} X_\alpha \to \prod_{\alpha\in A,\beta\in B} X^\beta_\alpha$ from Definition 2.3.18, i.e. all the morphisms in 2.2 are isomorphisms.

Proposition 2.3.32. *With the previous notations and with*

$$\Psi_1 = \left(\prod_{\beta\in B} \Phi^{-1} \circ \Xi^{\Phi(s)}_\beta\right) \qquad \Psi_2 = \prod_{\alpha\in A}\left(\prod_{\beta\in B}\left(\pi_\alpha \circ \Phi^{-1} \circ \Xi^{\Phi(s)}_\beta\right) \circ \Phi \circ \Xi^s_\alpha\right)$$

$\Psi \circ \Phi$ and Ψ_2 are isomorphisms of connected loop-free categories. Furthermore, for all $\alpha \in A$, $\beta \in B$, $\pi_\beta\Psi_1$ and $\pi_\alpha\Psi_2$ are isomorphisms.

Proof.

- Ψ_1 is an isomorphism. Indeed, by Corollary 2.3.28 $\Xi^{\Phi(s)}_\beta \colon Y_\beta \to Y^{\Phi(s)}_\beta$ is an isomorphism. By 2.3.30, so is $\Phi^{-1} \colon Y^{\Phi(s)}_\beta \to \prod_{\alpha\in A} X^\beta_\alpha$. Thus, as products of isomorphism, all arrows in the following diagram are isomorphisms.

$$\prod_{\beta\in B} Y_\beta \xrightarrow{\prod_{\beta\in B}\Xi^{\Phi(s)}_\beta} \prod_{\beta\in B} Y^{\Phi(s)}_\beta \xrightarrow{\prod_{\beta\in B}\Phi^{-1}} \prod_{\beta\in B}\prod_{\alpha\in A} X^\beta_\alpha$$

- Let us prove that Ψ_2 is an isomorphism. Indeed, by Corollary 2.3.28, resp. Proposition 2.3.30 resp. 2.3.31, the following functors are all isomorphisms:

$$X_\alpha \xrightarrow{\Xi^s_\alpha} X^s_\alpha \xrightarrow{\Phi} \prod_{\beta\in B} Y^\alpha_\beta \xrightarrow{\prod_{\beta\in B}\pi_\alpha\circ\Phi^{-1}\circ\Xi^{\Phi(s)}_\beta} \prod_{\beta\in B} X^\beta_\alpha$$

It follows that $\Psi_2 = \prod_{\lambda\in A}\left(\prod_{\beta\in B}\left(\pi_\alpha \circ \Phi^{-1} \circ \Xi_\beta^{\Phi(s)}\right) \circ \Phi \circ \Xi_\alpha^s\right)$ is an isomorphism by composition and product of isomorphisms. Furthermore, each $\pi_\alpha\Psi_2$ is an isomorphism

□

Now we are tasked with proving the commutativity of the Diagram 2.3.

Proposition 2.3.33. *Let $\mu \in B$, $f \in \prod_{\alpha\in A} X_\alpha^\mu = \prod_{\alpha\in A} \pi_\alpha\Phi^{-1}[Y_\mu^{\Phi(s)}]$, with Ψ_1 and Ψ_2 as defined in Proposition 2.3.32. Then for all $\beta \in B$ and $\alpha \in A$*

$$\pi_\beta\pi_\alpha(\gamma^{-1} \circ \Psi_1 \circ \Phi)(f) = \pi_\beta\pi_\alpha\Psi_2(f) = \begin{cases} f_\alpha & \text{if } \beta = \mu \\ s_\alpha & \text{otherwise} \end{cases}$$

Proof. By definition of γ, the equality of Proposition 2.3.33 above is equivalent to

$$(\Psi_1 \circ \Phi(f)_\beta)_\alpha = ((\gamma \circ \Psi_2(f))_\beta)_\alpha = \begin{cases} f_\alpha & \text{if } \beta = \mu \\ s_\alpha & \text{otherwise} \end{cases}$$

Let $f = (f_\alpha)_{\alpha\in A} \in \prod_{\lambda\in A} X_\lambda^\mu$. By Proposition 2.3.30, $\Phi(f) \in Y_\mu^{\Phi(s)}$ with $\Phi(f) = (\Phi(\mathrm{id}_s), \Phi(f)_\mu, \mu)$ and $\Phi(f)_\mu \in Y_\mu$.

- For $\Psi_1 \circ \Phi = \prod_{\beta\in B} \Phi^{-1} \circ \Xi_\beta^{\Phi(s)} \circ \pi_\beta\Phi$

$$\begin{aligned} \pi_\beta\Psi_1 \circ \Phi(f) &= \Phi^{-1} \circ \Xi_\beta^{\Phi(s)} \circ \Phi(f)_\beta \\ &= \Phi^{-1}(\mathrm{id}_{\Phi(s)}, \Phi(f)_\beta, \beta) && \text{Definition 2.3.20} \\ &= \Phi^{-1}(\mathrm{id}_{\Phi(s)}, (\mathrm{id}_{\Phi(s)}, \Phi(f)_\mu, \mu)_\beta, \beta) \\ \pi_\beta\Psi_1 \circ \Phi(f) &= \begin{cases} \Phi^{-1}(\mathrm{id}_{\Phi(s)}) = \mathrm{id}_s & \text{if } \mu \neq \beta \\ \Phi^{-1}(\mathrm{id}_{\Phi(s)}, \Phi(f)_\mu, \mu) = f & \text{if } \mu = \beta \end{cases} \end{aligned}$$

- For $\gamma \circ \Psi_2 = \prod_{\alpha\in A}\left(\prod_{\beta\in B} \pi_\alpha\Phi^{-1} \circ \Xi_\beta^{\Phi(s)}\right)(\Phi \circ \Xi_\alpha^s)$

  - $\beta \neq \mu$
    By definition, $\mathrm{id}_s = \Phi^{-1}\Phi(\mathrm{id}_s) \in \Phi^{-1}[Y_\mu^{\Phi(s)}] = \prod_{\alpha\in A} X_\alpha^\mu$, such that for all $\alpha \in A$, $\Xi_\alpha^s f_\alpha \in \prod_{\alpha\in A} X_\alpha^\mu$ i.e. $\Phi \circ \Xi_\alpha^s(f_\alpha) \in Y_\mu^{\Phi(s)}$ and thus for all $\beta \neq \mu$, $\pi_\beta\Phi \circ \Xi_\alpha^s(f_\alpha) = \Phi(\mathrm{id}_s)_\beta$.
    Thus, for all $\beta \neq \mu$

$$\begin{aligned} \pi_\alpha\pi_\beta\gamma \circ \Psi_2(f) &= \pi_\alpha\Phi^{-1}\Xi_\beta^{\Phi(s)}\left(\pi_\beta\Phi \circ \Xi_\alpha^s(f_\alpha)\right) \\ &= \pi_\alpha\Phi^{-1}\Xi_\beta^{\Phi(s)}(\Phi(\mathrm{id}_s)_\beta) \\ \pi_\alpha\pi_\beta\gamma \circ \Psi_2(f) &= \pi_\alpha\mathrm{id}_s \end{aligned}$$

– $\mu = \beta$ Then, $f \in \prod_{\lambda\in A} X_\lambda^\mu$ therefore $\Phi(f) = (\Phi(\mathrm{id}_s), \Phi(f)_\beta, \beta)$. It follows that

$$\forall g_\beta \in Y_\beta, (\Phi(f), g_\beta, \beta) = (\Phi(\mathrm{id}_s), g_\beta, \beta) \tag{2.8}$$

Furthermore $\pi_\alpha\Phi^{-1}\Phi(f) = \pi_\alpha\Phi^{-1}\Phi(\mathrm{id}_s, f_\alpha, \alpha) = f_\alpha$. Hence,

$$\begin{aligned} f_\alpha &= \pi_\alpha\Phi^{-1}(\Phi(f), (\Phi(\mathrm{id}_s, f_\alpha, \alpha))_\beta, \beta) && \text{By Corollary 2.3.26} \\ &= \pi_\alpha\Phi^{-1}(\Phi(\mathrm{id}_s), (\Phi(\mathrm{id}_s, f_\alpha, \alpha))_\beta, \beta) && \text{By Eq. (2.8)} \\ f_\alpha &= \pi_\alpha\pi_\beta\gamma \circ \Psi_2(f) && \text{By definition of } \Psi_2 \end{aligned}$$

. □

Now that commutativity of Diagram 2.3 is proven, we extend the commutativity along one of the $\lambda \in A$, thus proving the commutativity of the Diagram 2.2. First as explained in the outline, we will to prove the faithfulness of the restriction, which is less trivial in our case.

Lemma 2.3.34. *With the previous notation, γ: $\prod_{\beta\in B}\prod_{\alpha\in A} X_\alpha^\beta \to \prod_{\alpha\in A}\prod_{\beta\in B} X_\alpha^\beta$ the natural isomorphism and with $\widetilde{s_\lambda}$ the singleton category for a given $\lambda \in A$, the following restrictions are faithful functors.*

$$\begin{aligned} \pi_\lambda\gamma^{-1}\Psi_1 \circ \Phi|_{X_\lambda^s} \qquad & \pi_{\{\alpha\in A|\alpha\neq\lambda\}}\gamma^{-1}\Psi_1 \circ \Phi|_{\prod_{\alpha\neq\lambda} X_\alpha \times \widetilde{s_\lambda}} \\ \pi_\lambda\Psi_2|_{X_\lambda^s} \qquad & \pi_{\{\alpha\in A|\alpha\neq\lambda\}}\Psi_2|_{\prod_{\alpha\neq\lambda} X_\alpha \times \widetilde{s_\lambda}} \end{aligned}$$

Proof. • $\Psi_1 \circ \Phi = \left(\prod_{\beta\in B} \Phi^{-1} \circ \Xi_\beta^{\Phi(s)}\right) \circ \Phi$

Let $f = (f_\alpha)_{\alpha\in A} \in X_\lambda^s$, then $\pi_\alpha f = \pi_\alpha \mathrm{id}_s$ for all $\alpha \neq \lambda$. Thus, by Corollary 2.3.26, for all $\beta \in B$ and all $\alpha \neq \lambda$,

$$\pi_\alpha\Phi^{-1}(\Phi(\mathrm{id}_s), \Phi(f)_\beta, \beta) = \pi_\alpha \mathrm{id}_s$$

i.e. $\pi_\alpha\pi_\beta\Psi_1 \circ \Phi(f) = \pi_\alpha \mathrm{id}_s$. Thus

$$\begin{aligned} \Psi_1(f) \circ \Phi &= (\mathrm{id}_s, \left(\pi_\beta\Psi_1(f)\right)_\lambda, \lambda)_{\beta\in B} \\ &= (\Xi_\lambda^s(\pi_\lambda\pi_\beta\Psi_1(f)))_{\beta\in B} \\ &= \left(\prod_{\beta\in B} \Xi_\lambda^s\right)(\pi_\lambda\pi_\beta\Psi_1(f))_{\beta\in B} \\ \Psi_1 \circ \Phi(f) &= \left(\prod_{\beta\in B} \Xi_\lambda^s\right) \circ \left(\pi_\lambda\gamma^{-1}\Psi_1 \circ \Phi\right)(f) \end{aligned}$$

This is true for all $f \in X_\lambda^s$, thus:

$$\Psi_1 \circ \Phi|_{X_\lambda^s} = \left(\prod_{\beta\in B} \Xi_\lambda^s\right) \circ \left(\pi_\lambda\gamma^{-1}\Psi_1 \circ \Phi\right)|_{X_\lambda^s}$$

By faithfulness of $\Psi_1 \circ \Phi$ (Proposition 2.3.32) and Ξ_λ^s (Lemma 2.3.27), this implies $\pi_\lambda\gamma^{-1}\Psi_1 \circ \Phi|_{X_\lambda^s}$ faithful.

Now let $f \in (f_\alpha)_{\alpha\in A} \in \prod_{\alpha\neq\lambda} X_\alpha \times \widetilde{s_\lambda}$, then $\pi_\lambda f = \pi_\lambda \mathrm{id}_s$. By Corollary 2.3.26, this implies for all $\beta \in B$

$$\pi_\lambda \Phi^{-1}(\Phi(\mathrm{id}_s), \Phi(f)_\beta, \beta) = \pi_\lambda \mathrm{id}_s$$

i.e. $\pi_\lambda\pi_\beta\Psi_1 \circ \Phi(f) = \pi_\lambda \mathrm{id}_s$ such that

$$\begin{aligned}
\Psi_1 \circ \Phi(f) &= (\pi_\beta\Psi_1 \circ \Phi(f), \mathrm{id}_{s\lambda}, \lambda)_{\beta\in B} && \text{Corollary 2.3.26}\\
&= ((\pi_\alpha\pi_\beta\Psi_1 \circ \Phi(f))_{\alpha\in A}, \mathrm{id}_{s\lambda}, \lambda)_{\beta\in B}\\
\Psi_1 \circ \Phi(f) &= ((\pi_\beta\pi_\alpha\gamma^{-1}\Psi_1 \circ \Phi(f))_{\alpha\in A}, \mathrm{id}_{s\lambda}, \lambda)_{\beta\in B}
\end{aligned}$$

Faithfulness of $\Psi_1 \circ \Phi$ then implies the faithfulness of $((\pi_\beta\pi_\alpha\gamma^{-1}\Psi_1 \circ \Phi(.))_{\alpha\neq\lambda})_{\beta\in B}$, i.e. the faithfulness of $\pi_{\alpha\neq\lambda}\gamma^{-1}\Psi_1 \circ \Phi$ on the subcategory $\prod_{\alpha\neq\lambda} X_\alpha \times \widetilde{s_\lambda}$

- $\Psi_2 = \prod_{\alpha\in A}\left(\prod_{\beta\in B}\left(\pi_\alpha \circ \Phi^{-1} \circ \Xi_\beta^{\Phi(s)}\right) \circ \Phi \circ \Xi_\alpha^s\right)$
  By Proposition 2.3.32, each $\pi_\alpha\Psi_2$ is an isomorphism, thus a fortiori, the restriction $\pi_\alpha\Psi_2|_{X_\alpha^s}$ is faithful. Hence, $\pi_{\{\alpha\in A|\alpha\neq\lambda\}}\Psi_2|_{\prod_{\alpha\neq\lambda} X_\alpha\times\widetilde{s_\lambda}}$ is faithful as a product of faithful functors.

□

Proposition 2.3.35. *With the previous notation,* $\Psi_1 \circ \Phi = \gamma \circ \Psi_2$

Proof. Let $\Psi \in \{\Psi_2, \gamma^{-1} \circ \Psi_1 \circ \Phi\}$. Let $f_\lambda^\mu \in X_\lambda^\mu$. By Proposition 2.3.33, Ψ_2 and $\gamma^{-1} \circ \Psi_1 \circ \Phi$ are equal when we restrict to $\prod_{\alpha\in A} X_\alpha^\mu$. As explained above, we wish to extend this equality to the full domain. Proposition 2.3.33 also implies that for all $g_\mu \in \prod_{\alpha\in A} X_\alpha^\mu$, $\pi_\lambda\Psi(g^\mu, f_\lambda^\mu, \lambda) = \pi_\lambda\Psi(s, f_\lambda^\mu, \lambda)$. Thus, by Lemma 2.3.24, for all $f_\lambda \in X_\lambda$, for all $g_\mu \in \prod_{\alpha\in A} X_\alpha^\mu$

$$\pi_\lambda\Psi(g^\mu, f_\lambda, \lambda) = \pi_\lambda\Psi(s, f_\lambda, \lambda) = \pi_\lambda\Psi\Xi_\lambda^s(f_\lambda) \stackrel{def}{:=} \Psi_\lambda(f_\lambda) \tag{2.9}$$

We want to prove that for all $\lambda \in A$, $\mu \in B$, $g^\mu \in \prod_{\alpha\in A} X_\alpha^\mu$, $f_\lambda \in X_\lambda$, the projection on any $\alpha \neq \lambda$ of $\Psi(g^\mu, f_\alpha, \alpha)$ depends only on g^μ, i.e.

$$(g^\mu, f_\lambda, \lambda) = \Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda)$$

- For $\alpha \neq \lambda$, $\pi_\alpha\Psi^{-1}\Psi(g^\mu, f_\lambda, \lambda) = g_\alpha^\mu = \pi_\alpha\Psi^{-1}\Psi(g^\mu)$. Thus, by Corollary 2.3.26,

  $$\pi_\alpha\Psi^{-1}(\Psi(g^\mu), \Psi(g^\mu, f_\lambda, \lambda)_\lambda, \lambda) = g_\alpha^\mu$$

  By definition of $\Psi_\lambda(f_\lambda)$ (Eq. (2.9))

  $$\pi_\alpha\Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda) = g_\alpha^\mu \tag{2.10}$$

- For λ, By Eq. (2.10), $\Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda) \in (X_\alpha^\mu, X_\lambda, \lambda)$, such that by Eq. (2.9)

  $$\begin{aligned}
  \pi_\lambda\Psi\Xi_\lambda^s(\pi_\lambda\Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda)) &= \pi_\lambda\Psi\Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda)\\
  &= \Psi_\lambda(f_\lambda)\\
  \pi_\lambda\Psi\Xi_\lambda^s(\pi_\lambda\Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda)) &= \pi_\lambda\Psi\Xi_\lambda^s(f_\lambda) && \text{Eq. (2.9)}
  \end{aligned}$$

  By faithfulness of $\pi_\lambda\Psi\Xi_\lambda^s$ (Lemma 2.3.34),

  $$\pi_\lambda\Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda) = f_\lambda \tag{2.11}$$

By Eq. (2.10) and Eq. (2.11), we have $(g^\mu, f_\lambda, \lambda) = \Psi^{-1}(\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda)$ i.e.

$$\Psi(g^\mu, f_\lambda, \lambda) = (\Psi(g^\mu), \Psi_\lambda(f_\lambda), \lambda) \tag{2.12}$$

So far Eq. (2.12) only holds when $g^\mu \in \prod_{\alpha\in A} X^\mu_\alpha$. Let us prove that it is in fact valid for all $g \in \prod_{\alpha\in A} X_\alpha$, i.e. for all $\lambda \in A$ and for all $f_\lambda \in X_\lambda$

$$\Psi(g, f_\lambda, \lambda) = (\Psi(g), \Psi_\lambda(f_\lambda), \lambda)$$

Let $\rho \in A$, $\rho \neq \lambda$. By Eq. (2.12), for all $f_\lambda \in X_\lambda$, $\pi_\rho\Psi(g^\mu, f_\lambda, \lambda) = \pi_\rho\Psi(g^\mu)$. By Lemma 2.3.24 this implies, for all $g \in \prod_{\alpha\in A} X_\alpha$, for all $f_\lambda \in X_\lambda$

$$\pi_{\bar\lambda}\Psi(g, f_\lambda, \lambda) = \pi_{\bar\lambda}\Psi(g) \coloneqq \Psi_{\bar\lambda}(g) \tag{2.13}$$

where $\pi_{\bar\lambda}$ is the projection on $A \setminus \{\lambda\}$.
Thus, proving $\Psi(g, f_\lambda, \lambda) = (\Psi(g), \Psi_\lambda(f_\lambda), \lambda)$ is equivalent to proving,

$$(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda) = \Psi(g, f_\lambda, \lambda)$$

i.e.

$$\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda) = (g, f_\lambda, \lambda)$$

Let us look at the different projections

- On λ.
  $\pi_\lambda\Psi^{-1}(\Psi(g, f_\lambda, \lambda)) = \pi_\lambda\Psi^{-1}(\Psi\Xi^s_\lambda(f_\lambda)) = f_\lambda$ implies by Corollary 2.3.26

  $$\pi_\lambda\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \pi_\lambda\Psi\Xi^s_\lambda(f_\lambda), \lambda) = f_\lambda$$
  $$\pi_\lambda\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda) = f_\lambda \quad \text{Eq. (2.9)}$$

- On $\bar\lambda$
  By Eq. (2.13), for all $h \in \prod_{\alpha\in A} X_\alpha$, $\pi_{\bar\lambda}\Psi(h, \mathrm{id}_{s\lambda}, \lambda) = \pi_{\bar\lambda}\Psi(h)$. Applying this to the element $h = \Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda)$ we get

  $$\begin{aligned}\pi_{\bar\lambda}\Psi(\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda), \mathrm{id}_{s\lambda}, \lambda) &= \pi_{\bar\lambda}\Psi\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda)\\ &= \pi_{\bar\lambda}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda)\\ &= \pi_{\bar\lambda}\Psi(g, f_\lambda, \lambda)\\ \pi_{\bar\lambda}\Psi(\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda), \mathrm{id}_{s\lambda}, \lambda) &= \pi_{\bar\lambda}\Psi(g, \mathrm{id}_{s\lambda}, \lambda) \qquad \text{Eq. (2.13)}\end{aligned}$$

  By faithfulness of $\pi_{\bar\lambda}\Psi|_{(\prod_{\alpha\in A} X_\alpha, s\tilde\lambda, \lambda)}$ (Lemma 2.3.34),

  $$(\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda), \mathrm{id}_{s\lambda}, \lambda) = (g, \mathrm{id}_{s\lambda}, \lambda)$$
  $$\pi_{\bar\lambda}\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda)) = \pi_{\bar\lambda}g$$

Thus $\Psi^{-1}(\Psi(g, f_\lambda, \lambda), \Psi_\lambda(f_\lambda), \lambda) = (g, f_\lambda, \lambda)$ and as such, for all $f_\lambda \in \lambda$, for all $g \in \prod_{\alpha\in A} X_\alpha$,

$$\Psi(g, f_\lambda, \lambda) = (\Psi(g), \Psi_\lambda(f_\lambda), \lambda)$$

As this is true for any $f, g \in \prod_{\alpha \in A} X_\alpha$, and any λ, we get

$$\Psi(f) = (\Psi_\alpha(f_\alpha))_{\alpha \in A}$$

All that is left to do is to prove that $\Psi_\alpha(f_\alpha)$ is the same whether $\Psi = \Psi_2$ or $\Psi = \gamma^{-1} \circ \Psi_1 \circ \Phi$. By Eq. (2.9)

$$\Psi_\lambda(f_\lambda) = \pi_\lambda \Psi \Xi^s_\lambda(f_\lambda)$$

But, by definition

$$\begin{aligned}
\pi_\beta \pi_\lambda \Psi_2(\Xi^s_\lambda f_\lambda) &= \pi_\beta \pi_\lambda (\Phi^{-1} \circ \Xi^{\Phi(s)}_\beta \circ \pi_\beta \circ \Phi \circ \Xi^s_\lambda)(\pi_\lambda(\Xi^s_\lambda f_\lambda)) \\
&= \pi_\beta \pi_\lambda (\Phi^{-1} \circ \Xi^{\Phi(s)}_\beta \circ \pi_\beta \circ \Phi \circ \Xi^s_\lambda)(f_\lambda) && \text{Lemma 2.3.27} \\
&= \pi_\beta \pi_\lambda (\Phi^{-1} \Xi^{\Phi(s)}_\beta)(\pi_\beta \Phi)(\Xi^s_\lambda f_\lambda) \\
&= \pi_\beta \pi_\lambda \gamma^{-1} \circ \Psi_1 \circ \Phi(\Xi^s_\lambda f_\lambda) && \text{by Definition of } \Psi_1
\end{aligned}$$

This is true for all λ and β, thus $\Psi_2 = \gamma^{-1} \circ \Psi_1 \circ \Phi$

□

Finally, we have proven all the steps to prove our extension of Hashimoto's theorem.

Theorem 2.3.36. *Every connected loop-free category has the strict refinement property.*

Chapter 3
A syntactic model of programs

"I'm significant!…Screamed the dust speck."

– Bill Watterson, *Calvin & Hobbes*

Although topology offers a lot of interesting and powerful tools, programs are discrete objects, and we believe that such heavy machinery should not be necessary when studying them. In this chapter we present our result from [43] which introduces a new model of conservative programs. This model is based directly on the syntax of programs, and has as principal objective of making implementation of the tools introduced in the previous section Section 1.3 much more practical. We follow the presentation of the Section 1.3 to present our model and will translate all the different objects used in topological directed models according to the following table. Our main objective is to get back the powerful deadlock detection algorithms in our syntactic setting, which necessitates the compact representation by covers of the regions. Once again our first concern is to make these tools easier to implement and work with in practice and will influence many choices in the following sections.

Geometric semantics	**Syntactic semantics**
Directed topological spaces	Partially ordered sets
Pruned state space	Authorized positions
Points	Positions
Paths	Paths
n-cubes	(Syntactic) cubes
Cubical covers	(Syntactic) covers
Cubical regions	Support
Maximal cubical covers	Normal forms

In order to study and present this model, we first introduce a variant of the **PIMP** language we call **NPIMP** focusing purely on the concurrent nature of the programs considered. Making full use of the nature of the language **NPIMP**, and of the notion

of conservativity, we represent the state (and execution traces) as positions of a partially ordered set, corresponding to prefixes of executions of our program.

Then to efficiently manipulate and describe the set of authorized positions, we introduce an analogous notion of cubical covers (Section 3.2.1) for the syntactic semantics of our program, where a (potentially infinite) set of positions called a support is described by a set of cubes (defined for generic posets) containing said points. We show in Section 3.2.3 that these "syntactic cover", under reasonable assumptions, inherit the same boolean algebra structure that their topological equivalent possess. Using this structure, we define a canonical representative for the supports corresponding to actual "forbidden" and "authorized" region of programs, and explain how to implement its computation in Section 3.3.

These constructions allow us to implement the deadlock detection Algorithm 1.4.33 for the syntactic model in a fully automated tool *Sparkling*, which can be found at [42].

3.1 Syntactic semantics of concurrent programs

3.1.1 Positions in programs

As seen in the previous Chapter 1, for most of the models we introduced previously, reachability is undecidable, and many assumptions were made on the programs we consider in order to circumvent this difficulty (coherence of programs Definition 1.3.18, conservativity Definition 1.2.23) and focus on concurrency properties.

To further focus on the problems linked to the concurrent nature of our programs, we will remove arithmetic variables and boolean conditions altogether from **PIMP**. Indeed, the definitions for reachability, deadlocks, unsafe and doomed regions of Definition 1.3.58 does not take these into account, and we shall do the same.

In this chapter, we introduce a variant of the **PIMP** language: the non-deterministic **PIMP**, or **NPIMP** for short. **NPIMP** uses similar constructors and actions but has no arithmetic expressions, nor boolean conditions and no variables, thus focusing purely on the parallel structure of the programs and their resource consumption.

Definition 3.1.1. Let $\mathcal{R}$ be a fixed, finite set of resources. Let $a \in \mathcal{R}$. The language **NPIMP** is generated by the following syntactic expressions, defined by their grammar:

- the set $\mathcal{X}$ of actions:

$$A \quad ::= \quad \mathtt{P}_a \quad | \quad \mathtt{V}_a \quad | \quad \mathtt{skip} \quad | \quad \dots$$

- the set $\mathcal{C}$ of commands, or programs:

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

In this language, P^* replaces `while` b `do` P and P`+`Q replaces `if` b `then` P `else` Q. 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 in the same way that topological models do.

In a **NPIMP** program, we do not consider boolean condition or arithmetic variables. Thus, a state σ in our set of states Σ for a program in **NPIMP** should be a function associating to each resource its consumption.

Definition 3.1.2. Given a program P in **NPIMP**, we write $\Sigma = \mathbb{Z}^{\mathcal{R}}$ for the set of *states* of P, consisting of functions assigning an integer to each resource. The *initial state* σ_0 is the constant function equal to 0. The operational semantics of our programming language consists of two functions:

- $[\![-]\!]_{\mathcal{X}} : \mathcal{X} \to \Sigma \to \Sigma$, describing the evaluations of actions
- $[\![-]\!]_{\mathcal{C}} : \mathcal{C} \to \Sigma \to \Sigma$, describing the evaluations of commands or programs.

As we only are considering resources, the function $[\![-]\!]_{\mathcal{X}}$ is easy to define. $\mathtt{P}_a$ increases the consumption of a resource, while $\mathtt{V}_a$ diminishes it. All other actions are considered to have no effect.

Definition 3.1.3. Given a program P in **NPIMP**, the *evaluation* of the actions $\mathtt{P}_a$ and $\mathtt{V}_a$ are the functions $[\![\mathtt{P}_a]\!]_{\mathcal{X}}, [\![\mathtt{V}_a]\!]_{\mathcal{X}} : \Sigma \to \Sigma$ defined as follows:

$$[\![\mathtt{P}_a]\!]_{\mathcal{X}}(\sigma) = \delta_a^+(\sigma_r) \qquad\qquad [\![\mathtt{V}_a]\!]_{\mathcal{X}}(\sigma) = \delta_a^-(\sigma_r)$$

where

$$\delta_a^+(\sigma)(b) = \begin{cases} \sigma(a) + 1 & \text{if } b = a \\ \sigma(a) & \text{otherwise} \end{cases} \qquad \delta_a^+(\sigma)(b) = \begin{cases} \sigma(a) - 1 & \text{if } b = a \\ \sigma(a) & \text{otherwise} \end{cases}$$

Instead of generating the state space inductively as in Definition 1.3.30 or Definition 1.3.13, we will instead generate the set of "states" or *positions* of our program directly.

By restraining ourselves to conservative programs, as in Section 1.2.3, the consumption of resources of an execution will depend only on its endpoints. For **NPIMP** consumption and evaluation of an execution is the same, as there is only resources. Thus, for such programs, we can define the semantics of programs solely for their positions (states).

3.1.1.1 Positions of a program

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

Definition 3.1.4. The *pre-positions* p are generated by the following grammar, with $n \in \mathbb{N}$:

$$p, q \quad ::= \quad \bot \;\mid\; \top \;\mid\; p\,;q \;\mid\; p^n \;\mid\; p\mathtt{+}\varnothing \;\mid\; \varnothing\mathtt{+}q \;\mid\; p \parallel q$$

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 ($p\mathtt{+}\varnothing$, $\varnothing\mathtt{+}q$) and we are executing two programs in parallel ($p \parallel q$).

Remark 3.1.5. 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 pre-positions which are actual positions (states) 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 pre-position of the form $p;q$ with $q \neq \bot$ is a *position* only when p is $\top$. Similarly, in a conditional branching $P+Q$, we cannot execute both subprograms: that is why positions of $P+Q$ are of the form $p+\varnothing$ and $\varnothing+q$.

Remark 3.1.6. In [43], we had previously defined positions of $P+Q$ as $p+\bot$ and $\bot+q$. This was changed in order to make some proofs easier to understand in Section 4.2. This has no effect on the results of this section and all proofs have been suitably modified to take this change into account.

Definition 3.1.7. We write $P \vDash p$ to indicate that a pre-position p is a *position* of a program P, this predicate being defined inductively by the following rules:

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

$$\frac{}{P \vDash \top} \qquad \frac{Q \vDash q}{P;Q \vDash \top;q} \qquad \frac{Q \vDash q}{P+Q \vDash \varnothing+q} \qquad \frac{P \vDash p \qquad Q \vDash q}{P \parallel Q \vDash p \parallel q}$$

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

Remark 3.1.8. In the rest of this thesis, and contrary to what we did in [43], we will identify the positions $P \parallel Q \vDash \bot \parallel \bot$ and $P \parallel Q \vDash \bot$. This makes it easier to represent parallel composition of processes as multidimensional "cubes". This identification does not change the properties we are concerned about.

3.1.1.2 Operational semantics

We will now, as in Section 1.1.2, introduce an intermediate reduction relation to describe how our prefixes of executions (our positions, which are also our state) progress towards another. Here, we do not restrict the transitions to those which would lock resources more than their capacity or release unlocked resources as in Section 1.2.4. We will allow such paths for now and filter them out afterwards.

Definition 3.1.9. The *reduction* relation is defined inductively by

$$\frac{}{P{+}Q \vDash \bot \to \varnothing{+}\bot} \qquad \frac{Q \vDash q \to q'}{P{+}Q \vDash \varnothing{+}q \to \varnothing{+}q'} \qquad \frac{}{P{+}Q \vDash \varnothing{+}\top \to \top}$$

$$\frac{}{P{+}Q \vDash \bot \to \bot{+}\varnothing} \qquad \frac{P \vDash p \to p'}{P{+}Q \vDash p{+}\varnothing \to p'{+}\varnothing} \qquad \frac{}{P{+}Q \vDash \top{+}\varnothing \to \top}$$

$$\frac{P \vDash p \to p'}{P \parallel Q \vDash p \parallel \varnothing \to p'{+}\varnothing} \qquad \frac{Q \vDash q \to q'}{P \parallel Q \vDash p \parallel q \to p{+}q'}$$

$$\frac{}{P;Q \vDash \bot \to \bot;\bot} \qquad \frac{P \vDash p \to p'}{P;Q \vDash p;\bot \to p';\bot} \qquad \frac{Q \vDash q \to q'}{P;Q \vDash \top;q \to \top;q'} \qquad \frac{}{P;Q \vDash \top;\top \to \top}$$

$$\frac{}{P^* \vDash \bot \to \bot^0} \qquad \frac{P \vDash p \to p'}{P^* \vDash p^n \to p'^n} \qquad \frac{}{P^* \vDash \top^n \to \bot^{n+1}} \qquad \frac{}{P^* \vDash n^*\top \to \top}$$

$$\frac{P \vDash p \to p'}{P^* \vDash p^* \to p'^*}$$

$$\frac{A \in \mathcal{X}}{A \vDash \bot \to \top}$$

Remark 3.1.10. 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 lower left rule $A \vDash \bot \to \top$ which can be interpreted as executing an action A.

Furthermore, the relation is well-defined as proven by the following lemma. Combined with Lemma 3.1.20 this formalizes the fact that the positions in Definition 3.1.7 captures exactly the expected states of our program.

Lemma 3.1.11. *If $P \vDash p \to p'$ holds then both $P \vDash p$ and $P \vDash p'$ hold.*

Proof. This is proved by an easy induction on the derivation of the reduction. For instance, consider the case where the last rule is

$$\frac{P \vDash p \to p' \qquad Q \vDash q}{P \parallel Q \vDash p \parallel q \to p' \parallel q}$$

By induction hypothesis, we have a derivation of $P \vDash p \to p'$ and thus both $P \vDash p$ and $P \vDash p'$ hold. We can use the rules

$$\frac{P \vDash p \qquad Q \vDash q}{P \parallel Q \vDash p \parallel q} \qquad \frac{P \vDash p' \qquad Q \vDash q}{P \parallel Q \vDash p' \parallel q}$$

to show that both $P \parallel Q \vDash p \parallel q$ and $P \parallel Q \vDash p' \parallel q$ hold. Other cases are similar. □

We can define the consumption of a path π. Each action $\mathtt{P}_a$ or $\mathtt{V}_a$ modifies the consumption according to intuition and the operational semantics already defined in Definition 3.1.3.

First, we recall and adapt some definitions from Section 1.1.2:

Definition 3.1.12. Given a program P, the *state space* $\mathcal{G}_P$ of this program is the graph whose vertices are the positions of P (Definition 3.1.7) and edges are the reductions (Definition 3.1.9).

Definition 3.1.13. Given a program P, a *path* on $\mathcal{G}_P$ is a sequence of reduction $\pi = (P \vDash p_i \to p_{i+1})_{0 \leq i < n}$ also written $P \vDash \pi : p \to_0^* p_n$ or $\pi \colon p_0 \to^* p_n$. With the following constructions:

- Given two paths $\pi \colon p \to^* q$ and $\pi' \colon q \to^* p'$ we write $\pi' \cdot \pi \colon p \to^* p'$ for their concatenation.
- The empty path on a state p is written $\varepsilon_p : p \to^* p$.

Furthermore, we say that

- An execution trace π of P is a path $P \vDash \pi : \bot \to_P^* p$. When $p = \top_P$, i.e. $P \vDash \pi : \bot \to_P^* \top_P$, we way that π is *total*.
- An execution trace π of P is *maximal* when it cannot be extended.
- An execution is *elementary* when it consists of one reduction step.
- A position p is reachable when there exists an execution trace π with p as target i.e. $P \vDash \pi : \bot \to_P^* p$.

As customary, we also write $P \vDash p \to^* p'$ (or sometimes simply $p \to^* p'$) when there exists a path $P \vDash \pi : p \to^* p'$.

Definition 3.1.14. Given an execution $P \vDash \pi : p \to^* p'$, we write $[\![P \vDash \pi : p \to^* p']\!] : \Sigma \to \Sigma$ (or sometimes simply $[\![\pi]\!]$) for its *evaluation* or *consumption* of mutexes. This function is defined for elementary executions on each state $\sigma \in \Sigma$ by:

- For $a \in \mathcal{R}$, $[\![\mathtt{P}_a \vDash \pi : \bot \to \top]\!](\sigma) = [\![\mathtt{P}_a]\!]_{\mathcal{X}} = \delta_a^+(\sigma)$,

$$\delta_a^+(\sigma)(b) = \begin{cases} \sigma(a) + 1 & \text{if } b = a \\ \sigma(a) & \text{otherwise} \end{cases}$$

- For $a \in \mathcal{R}$, $[\![\mathtt{V}_a \vDash \pi : \bot \to \top]\!] = [\![\mathtt{V}_a]\!]_{\mathcal{X}} = \delta_a^+(\sigma)$,

$$\delta_a^+(\sigma)(b) = \begin{cases} \sigma(a) - 1 & \text{if } b = a \\ \sigma(a) & \text{otherwise} \end{cases}$$

- For each reduction π, different from the two above, deduced with a rule of Definition 4.1.22 with no premise we have $[\![\pi]\!](\sigma) = \sigma$.
- For each reduction π deduced with a rule of Definition 4.1.22 with one reduction π' as premise, we have $[\![\pi]\!] = [\![\pi']\!]$.

And extended as a morphism of the category of executions on the monoid of consumptions, i.e. $[\![\varepsilon]\!] = \mathrm{id}$ and $[\![\pi \cdot \pi']\!] = [\![\pi']\!] + [\![\pi]\!]$.

Definition 3.1.15. A program P is *conservative* when for any pair of paths $\pi, \tau \colon x \to y$ in G_P with same source and target we have for any state $\sigma \in \Sigma$ and any resource $a \in \mathcal{R}$:

$$[\![\pi]\!](\sigma)(a) = [\![\tau]\!](\sigma)(a)$$

Example 3.1.16. 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 $c \neq b$.

Definition 3.1.17. The *consumption* of a program P is the partial function $\Delta(P) : \mathcal{R} \to \mathbb{Z}$ defined by induction on P by

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

$$\Delta(P;Q) = \Delta(P \parallel Q) = \Delta(P) + \Delta(Q)$$
$$\Delta(\texttt{if } b \texttt{ then } P \texttt{ else } Q) = \Delta(P) \quad \text{if } \Delta(P) = \Delta(Q)$$
$$\Delta(\texttt{while } b \texttt{ do } P) = \underline{0} \quad \text{if } \Delta(P) = \underline{0}$$

where $\underline{0}$ is the constant function equal to 0 and δ_a the indicator function of a.

Proposition 3.1.18. *The function Δ is only partially defined on programs. A program P is conservative if and only if $\Delta(P)$ is well-defined.*

Proof. See [43, Proposition 3.4]. □

As in Section 1.2.3, the consumption of any path is only defined by its endpoints. Thus, we can define the *consumption* of mutexes directly on positions. Furthermore, for a program **NPIMP**, consumption and evaluation of a path are the same thing, as the states only contains information about the resources.

Definition 3.1.19. Given a conservative program P and a position p in $\mathcal{P}(P)$, we define the *consumption* of p, written $[\![p]\!] \colon \mathcal{R} \to \mathbb{Z}$ as follows:

$$[\![p]\!] = [\![\pi]\!](\sigma_0)$$

for some path $\pi \colon \bot_P \to p$ and σ_0 the initial state of P from Definition 3.1.2

The above definition is always well-defined as the consumption of a path in conservative program only depends on the endpoints and furthermore, such a path always exists $\pi \colon \bot_P \to p$. This second point is proven in the following lemma:

Lemma 3.1.20. *Every position p of P is reachable for the relation $\to^*$.*

Proof. See the proof of Proposition 3.1.26 □

Definition 3.1.21. We say that a position p is *valid* when for all resources $a \in \mathcal{R}$, $0 \leq [\![p]\!] < \kappa_a$. We say that an execution $\pi = (p_i \to p_{i+1})_{i \in I}$ is *valid* when all positions p_i it visits are valid.

Proposition 3.1.22. *A global execution π is valid if and only if every position p it visits is valid.*

Proof. [43, Lemma 5] □

A valid execution is namely compatible with the expected semantics of resources in the sense that

- no resource is taken more than its capacity κ_a (without having been released in between),
- no resource is released without having been taken first.

Definition 3.1.23. Given a program P, the *authorized region* is defined as the set of all valid position. Its complement is called the *forbidden region* and any position in the forbidden region is said to be *forbidden*.

The authorized region corresponds to the pruned state space of Section 1.3, but in our case we prefer not pruning the state formally, instead we designate the position as forbidden and try to find which "cubes" intersect or not the forbidden region. This choice is important to define a tractable definition of intervals/cubes on our state space.

Finally, we define $[\![-]\!]_{\mathcal{C}}$, by the consumption of any path $\pi\colon \bot_P \to \top_P$, or alternatively simply by the consumption of $\top_P$.

Definition 3.1.24. Given a program P in **NPIMP**, its *evaluation* $[\![P]\!]_{\mathcal{C}}\colon \Sigma \to \Sigma$ is defined as follows:

$$[\![P]\!]_{\mathcal{C}}(\sigma) = [\![\top_P]\!] + \sigma$$

When considering execution traces, this reduces to

$$[\![P]\!]_{\mathcal{C}}(\sigma_0) = [\![\top_P]\!]$$

It is easy to check that $[\![\mathtt{P}_a]\!]_{\mathcal{C}} = [\![\mathtt{P}_a]\!]_{\mathcal{X}}$, and $[\![\mathtt{V}_a]\!]_{\mathcal{C}} = [\![\mathtt{V}_a]\!]_{\mathcal{X}}$.

3.1.2 A partial order on positions

In this section we will show that the reduction relation $\to$ from Definition 3.1.9 induces a partial order on the set of positions (i.e. the state space) that is much more tractable for computations and proofs. This will allow us to define the syntactic semantics of our programs as simple posets. In this model, the partial order plays an analogous role to the notion of direction in the geometric semantics of Section 1.3.

Definition 3.1.25. We write $\leq$ for the smallest reflexive relation on the positions of P such that

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

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

for $m < n$.

Proposition 3.1.26. *Given two positions p and p' of P, we have $p \leq p'$ if and only if $p \to^* p'$.*

Proof. We prove by induction on the program P that, given positions p and p' of P, having the relation $p \leq p'$ is equivalent to having reductions $p \to^* p'$. We suppose that p and p' are both distinct from $\bot$ and $\top$ below (these cases are easily handled separately).

- A. Trivial.
- $Q+R$. Then $p = q+r$ and $p' = q'+r'$ for some $q, q' \in \mathcal{P}(Q) \cup \{\varnothing\}$ and $r, r' \in \mathcal{P}(R) \cup \varnothing$. We have $q+r \leq r'+r'$ if and only if
  - $q \leq q'$ and $r = r' = \varnothing$, or
  - $r \leq r'$ and $q = q' = \varnothing$.

  Both are treated the same way, so let us suppose $q \leq q'$ and $r = r' = \varnothing$. By induction hypothesis, we have that $q \leq q'$ is equivalent to $q \to^* q'$ and thus to $q+\varnothing \to^* q'+\varnothing$ by the rule
  $$\frac{Q \vDash q \to q'}{Q+R \vDash q+\varnothing \to q'+\varnothing}$$
- $Q \parallel R$. Then $p = p \parallel q$ and $p' = q' \parallel r'$ for some $q, q' \in \mathcal{P}(Q)$ and $r, r' \in \mathcal{P}(R)$. By the inference rules of Definition 3.1.25, we have that $p \parallel q \leq q' \parallel r'$ is equivalent to both $q \leq q'$ and $r \leq r'$ which, by induction hypothesis, is equivalent to $q \to^* q'$ and $r \to^* r'$ which, by the rules
  $$\frac{Q \vDash q \to q' \qquad R \vDash r}{Q \parallel R \vDash q \parallel r \to q' \parallel r} \qquad\qquad \frac{P \vDash p \qquad Q \vDash q \to q'}{P \parallel Q \vDash p \parallel q \to p \parallel q'}$$
  is equivalent to $q \parallel r \to^* q' \parallel r'$.
- $Q;R$. Then $p = q;\bot$ or $p = \top;r$, and $p' = q';\bot$ or $p'\top;r'$ for some $q, q' \in \mathcal{P}(Q)$ and $r, r' \in \mathcal{P}(R)$. By the inference rules of Definition 3.1.25, we simply have to prove that
  $$Q \vDash q \leq q' \text{ and } R \vDash r \leq r' \iff Q;R \vDash q;\bot \to^* q';\bot \to^* \top;\bot \to^* \top;r \to^* \top;r'$$
  By induction hypothesis $Q \vDash q \leq q'$ and $R \vDash r \leq r'$ is equivalent to $q \to^* q'$ and $r \to^* r'$ which, by the rules
  $$\frac{Q \vDash q \to q' \qquad R \vDash \bot}{Q;R \vDash q;\bot \to q';\bot} \qquad\qquad \frac{Q \vDash \top \qquad R \vDash r \to r'}{Q;R \vDash \bot;r \to \bot;r'}$$
  implies $q;\bot \to^* q';\bot$ and $\bot;r \to^* \bot;r'$. By extension (with $q' = \top$ and $r = \bot$), we get the equivalence, by concatenation.
- $P = Q^*$. Then $p = q^n$ and $p' = q'^m$ for some $q, q' \in \mathcal{P}(Q)$ and $n, n' \in \mathbb{N}$.
  - If $n = m$, we have by the inference rules of Definition 3.1.25, that $q^n \leq q'^n$ is equivalent to $q \leq q'$ which, by induction hypothesis, is equivalent to $q \to^* q'$ which, by the rule
    $$\frac{Q \vDash q \to q'}{Q^* \vDash q^n \to q'^n}$$
    is equivalent to $q^n \to^* q'^n$.

– If $n < m$, by the inference rules, $q \leq \top$ equivalent to $q^k \to^* \top^k$ and $\bot \leq q'$ equivalent to $\bot^k \to^* q'^k$. Thus, by concatenation of the paths :

$$\frac{Q \vDash q' \to^* \top}{Q^\star \vDash q^n \to^* \top^n} \qquad \frac{Q \vDash \bot \to^* \top}{Q^\star \vDash \bot^k \to^* \top^k} \text{ for } n \leq k \leq m \qquad \frac{Q \vDash \bot \to^* q'}{Q^\star \vDash \bot^m \to^* \to^* q'^m}$$

gives the equivalence.

By taking $p = p'$ we get the Lemma 3.1.20. The case where p or p' equals $\top$ or $\bot$ can also be deduced this way. □

Proposition 3.1.27. *The relation $\leq$ is a partial order on the set $\mathcal{P}(P)$ of positions of P.*

Proof. The reflexivity and transitivity of $\leq$ follows, by Proposition 3.1.26, from the reflexivity and transitivity of $\to^*$, which are satisfied by definition.

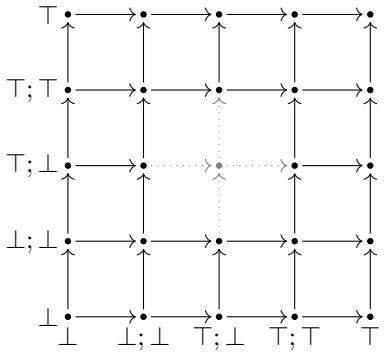
Let us show the antisymmetry of $\leq$. Given positions p and p' of P such that both $p \leq p'$ and $p' \leq p$ hold. The case where p or p' is either $\bot$ or $\top$ is immediate by definition of the order. For other cases, we reason by induction on P.

- A. By definition, $A \vDash p$ implies that p is either $\bot$ or $\top$ and this case is handled above.
- $R;Q$. The two positions are of the form $r;q$ and $r';q'$ such that, $r;q \leq r';q' \leq r;q$. By inference rules we have $r \leq r' \leq r$ and $q \leq q' \leq q$ which gives by induction, $r = r'$ and $q = q'$.
- $R \| Q$. Similar as above.
- $R+Q$. Similar as above.
- $Q^\star$. The positions are of the form r^n and q^m such that, $r^m \leq q^n \leq r^m$. By inference rules we have $n \leq m \leq n$. Thus, $m = n$ and $r \leq q \leq r$ by inference rules, which implies, by induction hypothesis $r = q$. □

Thus, we can define the *syntactic semantics* as the set of positions, equipped with the partial order $\leq$ on positions. By Proposition 4.1.32, two states are comparable, if there is an execution from one to the other.

Definition 3.1.28. The *syntactic semantics* associated to a program P is the partially ordered set $(\mathcal{P}(P), \leq_P)$ associated with a subset $\mathcal{P}_\mathcal{F}$ of its forbidden positions (as in Definition 3.1.23)

Example 3.1.29. Let us revisit the program $P = \mathtt{P}_a;\mathtt{V}_a \| \mathtt{P}_a;\mathtt{V}_a$ from Example 1.4.3. Its syntactic semantics is given below, with arrows indicating comparability. The central point, whose associated transitions have been dotted corresponds to a "forbidden position" of the program (one where the resource a has been taken more than its capacity). Contrary to previous models, we do not prune these positions from the semantics.



3.1.3 Syntactic semantics properties

The posets of positions of programs naturally exhibits some interesting properties. First of all incomparable elements are only generated by conditional branchings or parallelization of processes. As there is a finite amount of those, this means that every antichain of our syntactic semantics is finite. Furthermore, from any state/position of our programs, there can only have been a finite number of steps to reach this position, i.e. our semantics is well-founded. These properties combined imply that we are dealing with a special class of partial orders called well-orders [34]. These properties will be useful later on, when we will prove that our syntactic equivalent of cubical covers form a boolean algebra in Theorem 3.2.38.

Definition 3.1.30. Let $(X, \leq)$ a partially ordered set. We say that X is a *well-order* when one of the following equivalent properties hold

- for every infinite sequence $(x_i)_{i\in\mathbb{N}}$ of elements of X, there are indices $i < j$ such that $x_i \leq x_j$.
- X is well-founded and every antichain is finite.
- Every decreasing sequence $(x_i)_{i\in\mathbb{N}}$ of elements of X is stationary after a certain rank and every set of pairwise incomparable element is finite.

Proof. See [34, Basic Definitions and Tree Theorem] □

Proposition 3.1.31. *The poset $\mathcal{P}(P)$ 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 finite products, finite co-products and contain $(\mathbb{N}, \leq)$. A proof of this can be found in [48, Section 1.1.3].

Then, given a program P, we define $\mathcal{P}(P)^+ = \mathcal{P}(P) \setminus \{\bot, \top\}$. It is much easier to reason inductively on this set of positions, so we will first prove

$$\mathcal{P}(P) \text{ is a well-order if and only if } \mathcal{P}(P)^+ \text{ is a well-order}$$

Indeed both well-foundedness and finitude of antichains of these sets are equivalent:

- Given a decreasing sequence $(p_n)_{n\in\mathbb{N}} \in \mathcal{P}(P)^{\mathbb{N}}$. If there exists $m \in \mathbb{N}$ such that $p_m = \bot$ or for each $n \in \mathbb{N}, p_n = \top$, the sequence is trivially stationary after rank m.

- Given an antichain $A \in \mathcal{P}(P)^I$ indexed by some set I, we have that $|A| < \infty$ is equivalent to $|A \bigcap \mathcal{P}(P)^+| < \infty$.

By induction on the program P, we show that $\mathcal{P}(P)^+$ is a well-order.

- $P = A$. Since $\mathcal{P}(P) = \{\bot, \top\}$ is finite, it is a well-order.

- For all following cases:

  – $P = R \parallel Q$. Using the inference rules

  $$\frac{P \vDash p \qquad Q \vDash q}{P \parallel Q \vDash p \parallel q} \qquad\qquad \frac{p \leq p' \qquad q \leq q'}{p \parallel q \leq p' \parallel q'}$$

  we have $\mathcal{P}(P)^+ = \mathcal{P}(R) \parallel \mathcal{P}(Q)$ equivalent to $\mathcal{P}(R) \times \mathcal{P}(Q)$ with the product order.

  – $P = R + Q$. Using the inference rules

  $$\frac{P \vDash p}{P + Q \vDash p + \varnothing} \qquad \frac{Q \vDash q}{P + Q \vDash \varnothing + q} \qquad \frac{p \leq p'}{p + \varnothing \leq p' + \varnothing} \qquad \frac{q \leq q'}{\varnothing + q \leq \varnothing + q'}$$

  we have $\mathcal{P}(P)^+ = \mathcal{P}(R) + \varnothing \bigcup \varnothing + \mathcal{P}(Q)$ equivalent to $\mathcal{P}(R) \sqcup \mathcal{P}(Q)$ with the canonical disjoint order.

  – $P = R ; Q$. Using the inference rules

  $$\frac{P \vDash p}{P ; Q \vDash p ; \bot} \qquad \frac{Q \vDash q}{P ; Q \vDash \top ; q} \qquad \frac{p \leq p' \qquad q \leq q'}{p ; q \leq p' ; q'}$$

  we have $\mathcal{P}(P)^+ = (\mathcal{P}(R) ; \bot) \setminus \{\top ; \bot\} \bigcup \top ; \mathcal{P}(Q)$ is equivalent to $(\{0\} \times \mathcal{P}(R) \setminus \top) \bigcup (\{1\} \times \mathcal{P}(Q))$ equipped with the lexicographic order.

  – $P = R^*$. Using the inference rules

  $$\frac{P \vDash p}{P^* \vDash p^n} \qquad \frac{p \leq p'}{p^n \leq p'^n} \qquad \frac{m < n}{p^m \leq p'^n}$$

  By the inference rules, $\mathcal{P}(P)^+ = \{r^n \mid r \in \mathcal{P}(R), n \in \mathbb{N}\}$ is equivalent to $\mathbb{N} \times \mathcal{P}(R)$ equipped with the lexicographic order.

The posets $\mathcal{P}(R)$ and $\mathcal{P}(Q)$ are well-orders by induction hypothesis, $\{0, 1\}$ and $\mathbb{N}$ as well, and well-orders are closed by all the above operations [48, Section 1.1.3, 1.1..4].
Thus, $\mathcal{P}(P)^+$ is a well order. □

Proposition 3.1.32. *The partial order $\leq$ on $\mathcal{P}(P)$ is a bounded lattice, with $\bot$ and $\top$ as smallest and largest elements, with supremum being determined by*

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

for $m < n$. The infimum admits a similar description.

Proof. Given a program P, let us prove that for any two positions $P \vDash p_1, p_2$, $p_1 \vee p_2$ corresponds to the supremum $\sup(p_1, p_2)$ of p_1 and p_2 in the classical sense, i.e. $\sup(p_1, p_2) = \min\{p \in \mathcal{P}(P) \mid p \geq p_1, p \geq p_2\}$.

We first remark that by definition, $\top \vee p = \top$ and $\bot \vee p = p$ corresponds to the supremum of the positions. We will thus suppose both p_1 and p_2 distinct from $\bot$ and $\top$ for the rest of the proof. We prove all other cases by an induction on the program P.

- $R = A$. Trivial.
- $P = Q;R$. The positions are of the form $p_1 = q_1;r_1$ and $p_2 = q_2;r_2$.
  By induction hypothesis, for $i = 1, 2$, we have

  $$q_1 \vee q_2 \geq q_i \text{ and } r_1 \vee r_2 \geq r_i$$

  and for every position $q;r$ of P such that $q;r \geq p_i$ for $i = 1, 2$, we have

  $$q \geq q_1 \vee q_2 \text{ and } r \geq r_1 \vee r_2$$

  By the inference rules, we therefore have $q;r \geq (q_1 \vee q_2);(r_1 \vee r_2) \geq q_i;r_i$, which proves

  $$\sup(p_1, p_2) = (q_1;r_1) \vee (q_2;r_2) = (q_1 \vee q_2);(r_1 \vee r_2) = p_1 \vee p_2$$

  We still need to prove that $P \vDash (q_1 \vee q_2);(r_1 \vee r_2)$.

  - If $q_1 = q_2 = \top$ or $r_1 = r_2 = \bot$ the respective inference rules guarantee that $P \vDash p_1 \vee p_2$.
  - Otherwise, $q_1 = \top$ and $r_2 = \bot$. Then $p_1 \vee p_2 = (\top \vee q_2);(r_1 \vee \bot) = \top;r_1$ and $Q;R \vDash \top;r_1$ by inference rules.
- $P = Q \parallel R$. Similar to the case above.
- $P = Q+R$. We have $p_1 = q_1+r_1$ and $p_2 = q_2+r_2$
  - Suppose that $r_1 = r_2 = \varnothing$. By the inference rules, positions of P comparable with p_i are $p = q+\varnothing$ with q comparable to q_i or $p = \top$ (not the supremum since $\top+\varnothing$ is a smaller upper bound). Thus, similarly as above, the inference rules imply
    $$\sup(p_1, p_2) = (q_1 \vee q_2)+\varnothing = (q_1+\varnothing) \vee (q_2+\varnothing)$$
  - The case where $q_1 = q_2 = \varnothing$ is similar.

  - Otherwise, $p_1 = q_1 + \varnothing, p_2 = \varnothing + r_2$. By inference rules, $p \geq p_i$ for $i = 1, 2$ implies $p = \top$. Thus,
$$\sup(p_1, p_2) = \top = p_1 \vee p_2$$

- $P = Q^*$. We have $p_1 = q_1^{n_1}$ and $p_2 = q_2^{n_2}$.
  - If $n_1 > n_2$, then $p_1 > p_2$. Thus, $\sup(p_1, p_2) = p_1 = p_1 \vee p_2$.
  - Otherwise, suppose $n_1 = n_2 = n$. By induction, $q \geq q_i$ for $i = 1, 2$ implies $q \geq q_1 \vee q_2$ and thus by inference rules, $q^n \geq (q_1 \vee q_2)^n$. And by definition of the supremum, the inference rules imply $(q_1 \vee q_2)^n \geq q_i^n$. Thus
$$(q_1 \vee q_2)^n = \sup(q_1^{n_1}, q_2^{n_2})$$ □

3.2 The boolean algebra of cubical regions

3.2.1 Cubes and regions of posets

In this section we define our representation of the state-space of programs, replacing (maximal) cubical covers in geometric semantics of Section 1.4.1 by (normal) *syntactic covers*. Instead of considering cubes between pairs of points, we consider cubes between syntactic positions. This makes implementation easier, as we can rely on the inductive nature of their definition for the different operators (intersection, complement, …).

We show that, under mild assumptions, these regions satisfy the same fundamental properties as cubical covers (forming a boolean algebra, supporting the existence of canonical representatives, etc.) and provide explicit ways of computing corresponding canonical operations on syntactic covers.

3.2.1.1 Intervals

We will replace the n-cubes of the directed topological models by a notion of cube on partially ordered sets, simply defined as pairs of points in our posets which represents/covers the set of points in between them.

Definition 3.2.1. Let $(X, \leq)$ a poset. A *cube* (x, y) in this poset is a pair of elements such that $x \leq y$. We denote by $\mathcal{C}(X)$, the set of all cubes of X.

To differentiate the cube as an object and as the set of point it represents, in the same way we separate cubical regions and cubical covers in Section 1.4.1, we introduce the *support* operator, which associate to a cube the associated region.

Definition 3.2.2. Let $(X, \leq)$ a poset. Let $I = (x, y) \in \mathcal{C}(X)$. 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*.

When considering, $\vec{I}^n$ with the standard product order, then the set $[x_1 \times \ldots \times x_n, y_1 \times \ldots \times y_n]$ corresponds to the region $\prod_{i=1}^n [x_i, y_i]$. The associated cube corresponds to the associated n-cube.

Definition 3.2.3. Given a poset $\mathcal{P}$, we call (syntactic) *region* of $\mathcal{P}$ any subset $X \subseteq \mathcal{P}$

There is a natural ordering on cubes based on the inclusion order of the set of points they represent.

Definition 3.2.4. Let $(X, \leq)$ a poset. Let $I, J \in \mathcal{C}(X)$. We define the partial order relation $\subseteq$ on cubes as follows:

$$I \subseteq J \text{ if and only if } [I] \subseteq [J]$$

Remark 3.2.5. We write $z \in I$ instead of $z \in [I]$: we have $z \in (x, y)$ if and only if $x \leq z \leq y$.

3.2.1.2 Syntactic covers

Now the syntactic equivalent of cubical covers (resp. cubical regions) is naturally defined as the syntactic covers (resp. their support).

Definition 3.2.6. A *region* R of a poset X is a set of cubes of X. A cover is *finite*, when it contains a finite number of cubes. We write $\mathcal{R}(X) = \mathfrak{P}(X \times X)$ for the set of covers of X, where $\mathfrak{P}(-)$ denotes the power set of a given set.

Definition 3.2.7. The *support* of a cover R is the set $[R]$ defined as follows:

$$[R] = \bigcup_{I \in R} [I]$$

Two covers are *equivalent* when they have the same support.

Equivalent covers are the syntactic equivalent of cubical covers of the same cubical regions. Such covers are different ways of describing the same underlying set of points of the state space. Just as there exists a maximal cubical cover, we should be able to exhibit a "most compact" representation of a region in its equivalence class.

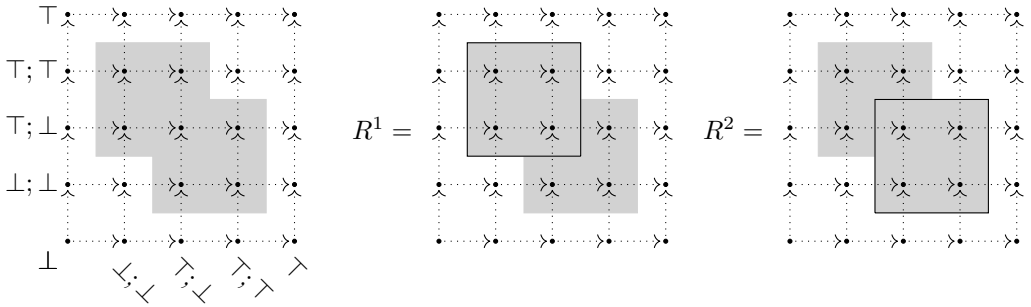
We recall that for a syntactic cube, its maximal cubical cover is given by the set of all maximal cubes $\mathcal{C}_n^{\max}(Y)$ included in the cubical region Y (Proposition 1.4.11). Intuitively, the "best" syntactic cover with a given support $Y \subseteq X$ will then consist of all the maximal cubes (w.r.t. $\subseteq$) contained in Y. We will call this region the *normal form* for a cover R and say that a cover is in normal form when it is equal to its own normal form.

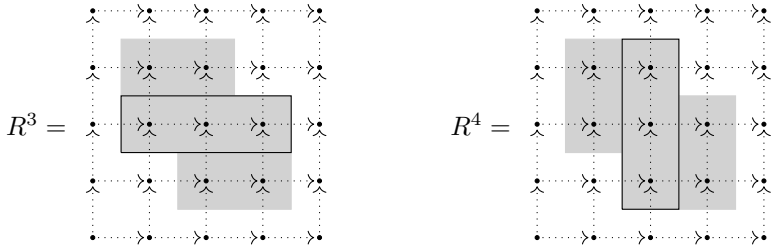
In order to formalize this, we re-introduce the (pre)order relation on n-cubes from Definition 1.4.8 and Definition 1.4.10.

Definition 3.2.8. Let R and S be two covers of X. The relation $\preceq$ defined as follows is a pre-order on $\mathcal{R}(X)$

$$R \preceq S \iff \forall R^i \in R, \exists S^j \in S, R^i \subseteq S^j$$

Example 3.2.9. Let us consider the program $P_1; P_2 \| P_1; P_2$. The set of its positions $\mathcal{P}(P)$ is represented below. On the left, we consider the region $X \subseteq \mathcal{P}(P)$ in grey, and various maximal cubes on this region.



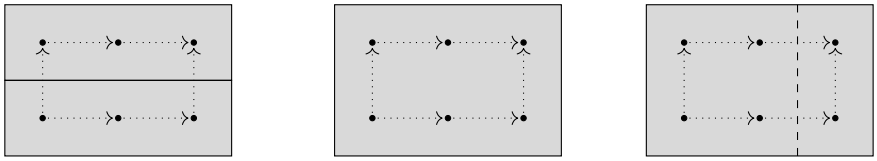


With

$$R^1 = (\top;\bot, \top;\top) \times (\bot;\bot, \top;\bot) \qquad R^2 = (\bot;\bot, \top;\bot) \times (\top;\bot, \top;\top)$$
$$R^3 = (\top;\bot, \top;\bot) \times (\bot;\bot, \top;\top) \qquad R^4 = (\bot;\bot, \top;\top) \times (\top;\bot, \top;\bot)$$

Both $R = \{R^1, R^2, R^3, R^4\}$ and $S = \{R^1, R^2\}$ are covers of X, but $S \preceq R$, as we consider that R carries more information about the region, as it has more of the maximal cubes included in X

Example 3.2.10. Given a program $P = P_1 + P_2$, let us consider the region $X = \mathcal{P}(P)$ as well as the three following covers of support X:



$R_1 = \{(\bot, \top + \varnothing), (\varnothing + \bot, \top)\}$ $R_2 = \{(\bot, \top)\}$ $R_3 = \{(\bot, \top), (\top + \varnothing, \top)\}$

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 cover 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, as we already explained in Section 1.4.2, there are many other maximal elements such as R_3 (or, in fact, any other cover obtained by adding cubes to R_2: the relation $\preceq$ is not antisymmetric). This motivates the introduction of the following refined order on covers, which is such that $R_1 < R_3 < R_2$:

Definition 3.2.11. We can define a partial order $\leq$ on the covers of a partial order X from the preorder $\preceq$ as follows. Given R, S two covers:

$$R \leq S \iff R \preceq S \preceq R \text{ and } S \subseteq R$$

When $R \preceq S$ and R and S are equivalent, we think of S as being a "more economical way" of describing the same support, because it uses bigger cubes: every cube of R is contained in one of S.

Lemma 3.2.12. *The relation $\leq$ is a partial order.*

Proof. Consider $R, S, T \in \mathcal{R}(X)$ such that $R \leq S \leq T$.

- Reflexivity. By reflexivity of $\preceq$ and $\subseteq$.
- Transitivity. Suppose $R \leq S$ and $S \leq T$. Thus, $R \preceq S \preceq T$, i.e. by transitivity of $\preceq$,
$$R \preceq T$$
Now suppose $T \preceq R$, then by transitivity $S \preceq R$ (resp. $T \preceq S$). By definition, $R \leq S$ (resp. $S \leq T$) implies $R \subseteq S$ (resp. $S \subseteq T$). Thus, by transitivity of $\subseteq$,
$$T \preceq R \text{ implies } R \subseteq T$$
- Antisymmetry. Suppose $R \leq S$ and $S \leq R$. This implies $R \preceq S \preceq R$. By definition of $\leq$, $R \leq S \leq R$ thus implies $R \subseteq S \subseteq R$. Hence, $R = S$. □

We give here two technical lemmas that will help us later on

Lemma 3.2.13. *The support function $[-] : \mathcal{R}(X) \to \mathfrak{P}(X)$ is increasing if we equip the first with $\leq$ and second with $\subseteq$ as partial orders.*

Proof. Given $R, S \in \mathcal{R}(X)$, such that $R \leq S$. For every $I \in R$ there exists $J_I \in S$ such that $I \subseteq J_I$, i.e. $[I] \subseteq [J_I]$. Then

$$[R] = \bigcup_{I \in R} [I] \subseteq \bigcup_{I \in R} [J_I] \subseteq \bigcup_{J \in S} [J] = [S]$$ □

Furthermore

Lemma 3.2.14. *The functions $[-] : \mathcal{R}(X) \to \mathfrak{P}(X)$ and $\mathcal{C} : \mathfrak{P}(X) \to \mathcal{R}(X)$ which associate to a region the set of all of its cubes are increasing and $[-]$ is left adjoint to $\mathcal{C}$. They form a Galois connection between the regions of X and the covers of X.*

Proof. We will prove the existence of an adjunction of posets (called a Galois connection) with $[-]$ the left adjoint to $\mathcal{C}$. First, we prove that both functions are increasing

- Given $R, S \in \mathcal{R}(X)$ such that $R \preceq S$, we have that for every $I \in R$ there exists $J_I \in S$ such that $I \subseteq J_I$, i.e. $[I] \subseteq [J_I]$ and thus
$$[R] = \bigcup_{I \in R} [I] \subseteq \bigcup_{I \in R} [J_I] \subseteq \bigcup_{J \in S} [J] = [S]$$
- Given $P, Q \in \mathfrak{P}(X)$ such that $P \subseteq Q$, we have that $(x, y) \in \mathcal{C}(P)$ is equivalent to $[x, y] \subseteq P \subseteq Q$ and thus $(x, y) \in \mathcal{C}(Q)$, which proves $\mathcal{C}(P) \subseteq \mathcal{C}(Q)$.

And by definition $R \in \mathcal{R}(X)$, such that $[R] \subseteq P$ is equivalent to $R \subseteq \mathcal{C}(P)$. This is explicitly the definition of a Galois connection. □

3.2.1.3 Normal form of regions

As in Section 1.4.2, we expect the "best description" of a subset of X by a cover to be given by a right adjoint $N : \mathfrak{P}(X) \to \mathcal{R}(X)$ to the support function $[-]$, with N associating to a subset Y of X the normal cover describing it.

The special adjoint functor theorem [38, V.8, Theorem 1] indicates that if such a right adjoint exists it should be defined as a particular limit as explained in Proposition 3.2.15 below:

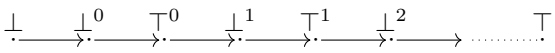
Proposition 3.2.15. *Given a poset X, if $N\colon \mathfrak{P}(X) \to \mathcal{R}(X)$ is the right adjoint of the functor $[-]\colon \mathcal{R}(X) \to \mathfrak{P}(X)$, it satisfies*

$$N(Y) = \max\{R \in \mathcal{R}(X) \mid [R] \subseteq Y\}$$

Proof. [38, V.8, Theorem 1] □

If all cubes of $\mathcal{C}(Y)$ are included in a maximal cube, then $N(Y)$ corresponds to the set of $\mathcal{C}^{\max}(Y)$ maximal cubes of Y. However, this condition is not always verified, and the adjoint N is not always well-defined, as illustrated in Example 3.2.16 below.

Example 3.2.16. 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 cover $N(Y)$ is well-defined. By the Lemma 3.2.18 below, $[N(Y)] = Y =]0,1[$. 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 cover R obtained from $N(Y)$ by removing this cube and replacing it by $[0, 1-\eta/2]$ is such that $[R] = [N(Y)] = Y$ and $N(Y) < R$, contradicting Proposition 3.2.15. A very similar situation can be observed in the case of programs, by considering the program $P = A^\star$ for some action A. We write $X = \mathcal{P}(P)$ for its poset of positions:



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

Remark 3.2.17. In order to accommodate with situations such as in previous example, one could think of allowing (semi-)open cubes in addition to closed ones in covers. This would make it so that N is always well-defined. However, the operations on those quickly become very difficult to handle because it turns out that, in the case of programs of the form $P \,\|\, Q$, we need to be able to specify whether bounds of cubes are open or not for each component of the parallel composition [30, 9].

The *normal form* of a region R is, when defined, the cover $N([R])$, although for convenience's sake we sometimes simply write $N(R)$ (justified by the Lemma 3.2.18), where N is defined by the formula of Proposition 3.2.15 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 3.2.18. *Given $Y \subseteq X$ such that $N(Y)$ is defined, one has $[N(Y)] = Y$.*

Proof. Given $Y \subseteq X$ such that $N(Y)$ is defined. The definition of a left adjoint states that for each $R \in \mathcal{R}(X)$, we have $[R] \subseteq Y$ if and only if $R \leq N(Y)$. By reflexivity of $\leq$, $N(Y) \leq N(Y)$ holds, so

$$[N(Y)] \subseteq Y$$

Furthermore $[-]$ increasing implies

$$Y = [\mathcal{C}(Y)] \subseteq [N(Y)]$$

Thus $N(Y) = Y$. □

In the general case, for a partial order X, it is not easy to characterize exactly the supports $Y \subseteq X$ that admit a normal form $N(Y)$. Thankfully, as we will be considering state spaces of programs and not any posets, we can impose further restrictions. First, we have seen that the syntactic semantics of a program is a well order Proposition 3.1.31. Sadly, as shown in Example 3.2.16 this is not enough to ensure that all supports will have a normal form. What we do know is that, usually getting a cover of the forbidden region of a program is much easier than getting a cover of the authorized region directly. In Section 1.4.2, and especially for the Algorithm 1.4.33, the authorized region is computed (in normal form) as the complement of the forbidden region, which is naturally obtained in normal form. This suggests that the correct supports to be considering are the subsets $Y \subseteq X$ that admit a normal form and whose complement $Y^{\mathrm{c}} = X \setminus Y$ also admits a normal form. Formally:

Definition 3.2.19. Suppose given a partial order X. We define the set $\mathcal{N}(X)$ of *normal supports* as follows:

$$\mathcal{N}(X) = \{Y \subseteq X \mid \text{both } N(Y) \text{ and } N(Y^{\mathrm{c}}) \text{ exists and are finite}\}$$

Thankfully, regions of $\mathcal{N}(X)$ are much easier to work with. Indeed, we will prove in Proposition 3.2.37 that, given suitable restrictions on X, these supports can be characterized as supports of a cover where all cubes admit a normalizable complement. This helps a lot as it is easy to determine if a cube admits such a complement Proposition 3.2.26. Furthermore, we will show that under reasonable assumptions the set $\mathcal{N}(X)$ forms a boolean algebra, similarly to the cubical covers of Section 1.4.

Remark 3.2.20. When only considering programs, we could have tried to prove that the only regions that are not normalizable do not correspond to authorized regions of conservative programs instead and proceed from there, but we have favoured a more general approach, which is equivalent in this case. Our previous claim is detailed later in the proof of Proposition 3.3.3.

3.2.2 Finitely complemented regions

As explained before, for our practical applications, given a cover R in $\mathcal{R}(X)$, we need to be able to compute a cover $\overline{R}$ which covers the complement of the cover, i.e. a cover $\overline{R}$ such that $[\overline{R}] = \overline{[R]}$. Moreover, in Section 1.4.2, we are able to compute the normal form of the complement, even when the cover is not in normal form. This suggests that for a cover of an element of $\mathcal{N}(X)$, we can compute their normal form $N([R])$ as the bi-orthogonal construction $\overline{\overline{R}}$.

In order to do this, we will first prove that we can define intersection and union operators for cover with normal supports compatible with the standard operations on the supports and conserving the property of having a normal support. In more specific terms, we will

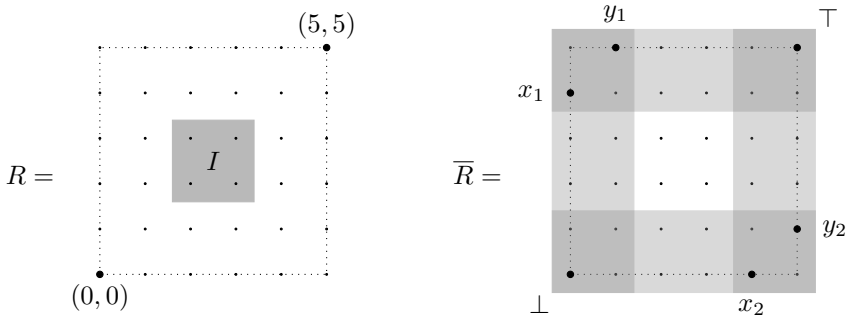
show that – under suitable hypothesis – regions which admit a normal form (and whose complement admit a normal form) have a boolean algebra structure, providing explicit algorithmic constructions for the corresponding operations.

This generalizes the situation considered in [18] (which is limited to regions which are subsets of $\mathbb{R}^n$) and in [30] (which is limited to products of graphs). In order to ensure that our constructions are applicable in practice, we only consider covers 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.

3.2.2.1 The complement of a cube

We begin by remarking that a cover $R = \{I\}$, consisting of single cube, is already in normal form (i.e. $N(R) = R$). Thus, $R \in \mathcal{N}(X)$ if and only if $[I]^c$ has a normal form. Hence, in order to characterize regions of $\mathcal{N}(X)$ we will first begin by investigating the covers corresponding to complements of a single cube.

As an example, let us 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 below:



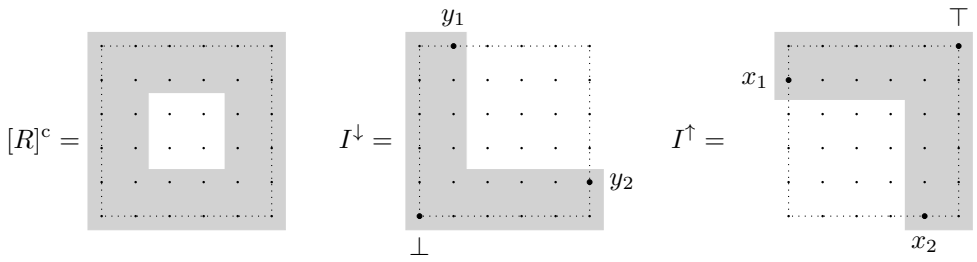
Then with:

$$\begin{array}{lll} \bot = (0,0) & y_1 = (1,5) & y_2 = (5,1) \\ \top = (5,5) & x_1 = (0,4) & x_2 = (4,0) \end{array}$$

The normal form of the complement $[I]^c$ of I is the cover

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

Now if we define $I^{\downarrow}$ (resp. $I^{\uparrow}$) is the set of elements of X which are not above $(2,2)$ (resp. below $(3,3)$). Then, as shown in the figure below, $[R]^c = I^{\downarrow} \cup I^{\uparrow}$.



Moreover, the set $I^\downarrow$ (resp. $I^\uparrow$) corresponds to all the points lower than an element of the finite set $\{y_1, y_2\}$ (resp. $\{x_1, x_2\}$).

We will actually see that, the set $I^\downarrow$ (resp. $I^\uparrow$) having a maximal (resp. minimal) antichain is actually the key to the existence of the normal form of the complement, and in that case we even have a description of the normal form (Proposition 3.2.26). First we need to introduce a few definitions before we can formalize this characterization.

Definition 3.2.21. Given a set $Y \subseteq X$, its *lower closure* $\downarrow Y$ and *lower complement* $Y^\downarrow$ are respectively the sets

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

The *upper closure* $\uparrow Y$ and *upper complement* $Y^\downarrow$ of Y are defined dually:

$$\uparrow Y = \{x \in X \mid \exists y \in Y.\ y \leq x\} \qquad Y^\uparrow = (\downarrow Y)^c = \{x \in X \mid \forall y \in Y. x \not\leq y\}$$

Definition 3.2.22. Given a set Y we write:

- $\max Y = \{y \in Y \mid \forall z \in y, z \not> y\}$ for the set of maximal elements of Y
- $\min Y = \{y \in Y \mid \forall z \in y, z \not< y\}$ for the set of minimal elements of Y

Definition 3.2.23. Given a set $Y \subseteq X$ we say that

- Y is *finitely lower generated* when there exists a finite set Y' such that $Y = \downarrow Y'$
- Y is *finitely upper generated* when there exists a finite set Y' such that $Y = \uparrow Y'$

By extension, we say that an element x of X is finitely lower (resp. upper) generated when $\{x\}$ is.

Lemma 3.2.24. *If Y is finitely lower generated then $\max Y$ is finite and, we have $Y = \downarrow \max Y$.*

Proof. Given a finitely lower generated poset $(Y, \leq)$. There exists a finite set $X \subseteq Y$ such that $Y = \downarrow X$. We want to prove that, $\max Y = \max X$ and that it is a lower generator of Y. By finiteness of X, any strictly increasing sequence $(z_i)_{i \in \mathbb{N}} \in X^{\mathbb{N}}$, is finite. This implies

$$\downarrow \max X = \downarrow X \setminus \{z \in X \mid \exists z' \in X, z < z'\} = \downarrow X = Y$$

Given an element $y \in \max Y \subseteq Y = \downarrow \max X$, there exists $z \in \max X \subseteq Y$ such that $y \leq z$. Maximality w.r.t. inclusion of elements of $\max Y$ implies $z = y$. Thus,

$$\max Y \subseteq \max X$$

Now given an element $z \in \max X$ and $y \in Y$ such that $z < y$. By definition, $\downarrow \max X = Y$ implies the existence of $z' \in \max X$ such that $z < y \leq z'$. This contradicts the definition of $\max X$. Thus, for all $y \in Y, z \not< y$, i.e. $z \in \max Y$. And thus

$$\max X \subseteq \max Y$$

Thus $\max Y = \max X$, finite and $\downarrow \max Y = \downarrow \max X = \downarrow X = Y$ □

Definition 3.2.25. Given a set $Y \subseteq X$, We say that Y is

- *finitely lower complemented* when $Y^{\downarrow}$ is finitely lower generated i.e. there exists Y' a finite set such that $(\downarrow Y)^{\mathrm{c}} = \downarrow Y'$
- *finitely upper complemented* when $Y^{\uparrow}$ is finitely upper generated i.e. there exists Y' a finite set such that $(\uparrow Y)^{\mathrm{c}} = \uparrow Y'$
- *finitely complemented* when it is both finitely lower and upper complemented.

By extension, we say that an element x of X is finitely lower (resp. upper) complemented when $\{x\}$ is.

From this definition we can characterize the covers $R = \{I\}$ corresponding to a single cube, which are in $\mathcal{N}(X)$. Indeed, such a region $[R]$ is in $\mathcal{N}(X)$ if and only if $[I]$ is finitely complemented.

Proposition 3.2.26. *Given a bounded lattice X and I a cube, the cover $R = \{I\}$ has a complement in normal form if and only if $[I]$ is finitely complemented. In this case, the normal form of its complement is*

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

Proof. Suppose given a poset $(X, \leq)$, $I = (s, t)$ a cube of X, and consider the cover $R = \{I\}$.

Left-to-right implication. Suppose that R has a complement in normal form, i.e. we have

$$\overline{[R]} = [N(\overline{[R]})]$$

where $N(\overline{[R]})$ is a finite cover. We have to show that I is finitely complemented, which means that the lower complement satisfies

$$I^{\downarrow} = \downarrow \max(I^{\downarrow})$$

with $\max(I^{\downarrow})$ finite, and dually for complement (we only handle the case of the lower complement here, the property for the upper complement being similar). By Lemma 3.2.24, it is enough to show that $I^{\downarrow}$ is finitely generated. We show here that it is generated by the set

$$G = \{x \in X \mid (\bot, x) \in N(\overline{[R]})\}$$

i.e.

$$I^{\downarrow} = {\downarrow} G$$

where G is finite because $N(\overline{[R]})$ is supposed to be finite. We show the equality by showing both inclusions.

- $I^{\downarrow} \subseteq {\downarrow} G$. Suppose given $x \in I^{\downarrow}$. Since $I^{\downarrow}$ is downward closed, we have $(\bot, x) \in \mathcal{C}(I^{\downarrow})$ and

$$\begin{aligned}
\mathcal{C}(I^{\downarrow}) &\leq \mathcal{C}(\overline{[R]}) && \text{because } I^{\downarrow} \subseteq \overline{[R]} \text{ and } \mathcal{C} \text{ is increasing} \\
&= \mathcal{C}([N(\overline{[R]})]) && \text{by hypothesis} \\
&\leq N(\overline{[R]}) && \text{by (3.2.15), because } [\mathcal{C}([N(\overline{[R]})]] = [N(\overline{[R]})] = \overline{[R]}
\end{aligned}$$

  This means that $(\bot, x) \subseteq (a, b)$ for some element $(a, b) \in N(\overline{[R]})$. And thus $a = \bot$ and $x \leq b$ with $b \in {\downarrow} G$.

- ${\downarrow} G \subseteq I^{\downarrow}$. Since $I^{\downarrow}$ is downward closed, it is enough to show $G \subseteq I^{\downarrow}$. Fix $x \in G$. We have $(\bot, x) \in N(\overline{[R]})$. In order to show $x \in I^{\downarrow}$, we have to show that $x \not\geq s$. Namely, if $x \geq s$, we would have $[\bot, x] \cap [I] \neq \emptyset$, which contradicts the hypothesis that $(\bot, x) \in N(\overline{[R]})$ and thus $[\bot, x] \subseteq \overline{[R]}$.

Right-to-left implication. Suppose that $[I]$ is finitely complemented, i.e. both $I^{\downarrow}$ and $I^{\uparrow}$ are finitely generated. Let us show that the normal form of $\overline{[R]}$ is

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

We write

$$S = \{[\bot, y] \mid y \in \max(I^{\downarrow})\} \qquad T = \{[x, \top] \mid x \in \min(I^{\uparrow})\}$$

We first remark that

$$\overline{[R]} = I^{\downarrow} \cup I^{\uparrow} = [S] \cup [T] = [S \cup T]$$

Namely, we have

$$\begin{aligned}
I^{\downarrow} &= {\downarrow} \max(I^{\downarrow}) && \text{by Lemma 3.2.24 since } I \text{ supposed to be finitely lower complemented} \\
&= [S]
\end{aligned}$$

and similarly $I^{\uparrow} = [T]$. Therefore, $\overline{R}$ is a cover whose support is $\overline{[R]}$, and it remains to be shown that the cover $\overline{R}$ is in normal form, i.e. $\overline{R} = N(\overline{R})$. We then proceed by using the definition of $N(\overline{R})$. Given a cover U such that $[U] \subseteq \overline{[R]}$, let us show that $[U] \leq \overline{R}$.

- $U \preceq \overline{R}$. By Lemma 3.2.14, we have $U \preceq \mathcal{C}([U]) \preceq \mathcal{C}(\overline{[R]}))$, and it suffices to show that $\mathcal{C}(\overline{[R]}) \preceq \overline{R}$. We first show that we can express $\mathcal{C}(\overline{[R]}) = \mathcal{C}(\overline{[I]}) = \mathcal{C}(\overline{[s,t]})$ as

$$\mathcal{C}(\overline{[s,t]}) = \mathcal{C}(s^{\downarrow}) \cup \mathcal{C}(t^{\uparrow})$$

  By definition of the complement, we have that $\overline{[s,t]} = s^{\downarrow} \cup t^{\uparrow}$. Therefore,

$$\begin{aligned}
\mathcal{C}(\overline{[I]}) =& \mathcal{C}(I^{\downarrow} \cup I^{\uparrow}) \\
=& \mathcal{C}(I^{\downarrow}) \cup \mathcal{C}(I^{\uparrow}) \\
& \cup \{(a, b) \mid a \in s^{\downarrow} \setminus t^{\uparrow} \text{ and } b \in t^{\uparrow} \setminus s^{\downarrow}\} \\
& \cup \{(a, b) \mid a \in t^{\uparrow} \setminus s^{\downarrow} \text{ and } b \in s^{\downarrow} \setminus t^{\uparrow}\}
\end{aligned}$$

Given $(a,b) \in \{(a,b) \mid a \in s^{\downarrow} \setminus t^{\uparrow} \text{ and } b \in t^{\uparrow} \setminus s^{\downarrow}\}$, we have $a \leq t, s \leq b$, thus $a \vee s \leq b \wedge t$, and therefore $[a,b] \cap [s,t] = [a \vee s, b \wedge t] \neq \emptyset$, and thus $(a,b) \notin \mathcal{C}(\overline{[I]})$. Similarly, we have $\{(a,b) \mid a \in t^{\uparrow} \setminus s^{\downarrow} \text{ and } b \in s^{\downarrow} \setminus t^{\uparrow}\} \cap \mathcal{C}(\overline{[I]}) = \emptyset$, from which we conclude

$$\mathcal{C}(\overline{[I]}) = \mathcal{C}(I^{\downarrow}) \cup \mathcal{C}(I^{\uparrow})$$

Now we show that $\mathcal{C}(I^{\downarrow}) \preceq S$. Given $(i,j) \in \mathcal{C}(I^{\downarrow})$. By definition of S, there exists $y \in \max I^{\downarrow}$ such that $i \leq j \leq y$ i.e. $(i,j) \subseteq (\bot, y) \in S$. Thus, $\mathcal{C}(I^{\downarrow}) \preceq S$. Similarly, $\mathcal{C}(I^{\uparrow}) \preceq T$. Thus,

$$\mathcal{C}(\overline{[R]}) = \mathcal{C}(I^{\downarrow}) \cup \mathcal{C}(I^{\uparrow}) \preceq S \cup T = \overline{R}$$

- $\overline{R} \preceq U$ implies $\overline{R} \subseteq U$. We show that under those condition we have $S \subseteq U$. Given $(\bot, x) \in S$, $\overline{R} \preceq U$ trivially implies $S \preceq U$, i.e. there exists $[u,v] \in U, [\bot, x] \subseteq [u,v]$. Immediately, we get $u = \bot$ and $v \in I^{\downarrow}$ (indeed $v \geq s$ implies $[\overline{I}] \cap I \supseteq \{(\bot, v)\} \cap I \neq \emptyset$). Furthermore, by definition of S, for $v \in I^{\downarrow}$, there exists $y \in \max I^{\downarrow}$ such that $(\bot, v) \subseteq (\bot, y)$. Finally, $(\bot, y)$ is a cube of S, such that

$$S \ni (\bot, x) \subseteq (u,v) = (\bot, v) \subseteq (\bot, y) \in S$$

By definition of $\max I^{\downarrow}$ this implies $(\bot, x) = (u,v) = (\bot, y)$. Thus,

$$S \subseteq U$$

By the same reasoning we get $T \subseteq U$. Thus, $\overline{R} \preceq U$ implies $\overline{R} \subseteq U$ □

3.2.2.2 The complement of a cover

We now explain how to construct the complement of a cover by using the previous complement of a cube. Then, the complement of the cover will be defined as the intersection of all the complements of its cubes. Leveraging the fact that for a single cube, the complement is in normal form, we will also show that the complement of a cover will contain all maximal cubes of its support.

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 3.1.31, is a reasonable restriction for the applications we have in mind. Our aim is now to generalize Proposition 3.2.26, and characterize normalizable covers (as opposed to cubes) which admit a complement in normal form. As we will show in Proposition 3.2.37, under suitable restrictions on X, normal supports coincide with the subset of X corresponding to a cover containing only finitely complemented cubes.

Definition 3.2.27. Given a poset X and a cover R on X. We say that R is *finitely complemented* if it contains only cubes whose support are finitely complemented in the sense of Definition 3.2.25. We write $\mathcal{R}_{\mathcal{F}}(X)$ for the set of finitely complemented covers of X, defined as:

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

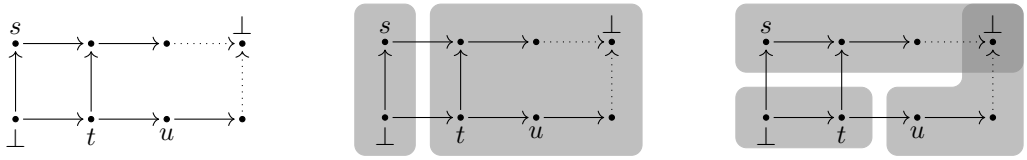
And $\mathcal{F}(X)$ the set whose elements are the supports of covers in $\mathcal{R}_{\mathcal{F}}(X)$:

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

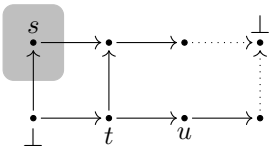
Remark 3.2.28. We bring attention to the reader that this does not state that the support of the cover should be finitely complemented.

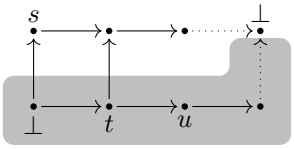
In order to apply Algorithm 1.4.33, we need to be able to perform the intersection, union and complement of the cubical covers. In order to efficiently manipulate these intermediary covers, we will need to be able to normalize them, and thus guarantee that they do have a normal form. Thus, we will need to prove that $\mathcal{N}(X)$ forms a boolean Algebra. This is not always the case as shown in Example 3.2.29 below.

Example 3.2.29. Let us construct an example of a well-ordered, bounded lattice, where $\mathcal{N}(X)$ is not a boolean algebra. Consider the space X, on the left below, where the arrows indicate comparability w.r.t. the order on X, and dotted arrows indicate an infinite countable chain of ordered points. It is easy to show that X verifies the required properties. Now let us consider the cubes $(\bot, s)$ and $(\bot, t)$. As shown on the right below, they are both normal supports i.e. in $\mathcal{N}(X)$.



Then, if we consider the intersection $[\{(\bot, s)\}] \cap [\overline{\{(\bot, t)\}}]$, we see that it is equal to the single point s but, the complement of (s, s) does not have a normal form: Indeed, as in Example 3.2.16 should we consider cubes of the form in the figure on the right below, We can show that there is no maximal element.

$[\bot, s] \cap [\bot, t]^{\mathrm{c}} =$ 

$s^{\downarrow} =$ 

This implies that (s, s) is not in $\mathcal{N}(X)$ and as such $\mathcal{N}(X)$ is not a boolean algebra.

The condition which emerged in order to capture such situations is the following one:

Definition 3.2.30. 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 3.2.31. In the case where X is a well-order, any 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.

In the next section we will show the following theorem, justifying the restriction we imposed on X

Theorem 3.2.38. *Given a well-ordered bounded lattice X:*

The poset $\mathcal{N}(X)$ is a boolean algebra if and only if X is finitely complemented

And furthermore, we will show that the set of normal support is, in fact, the set of finitely complemented ones.

Proposition 3.2.37. *Given a well-ordered bounded finitely complemented lattice X, the finitely complemented supports coincide with normal ones, i.e. we have*

$$\mathcal{F}(X) = \mathcal{N}(X)$$

3.2.3 Boolean algebra of finitely complemented regions

In this section, we will prove that finitely complemented supports (and thus normal supports) form a boolean algebra (Theorem 3.2.38). We will also show that this extends to finitely complemented covers and normalizable covers.

The plan of our proof for Theorem 3.2.38 is as follows: we first show that $\mathcal{F}(X)$ forms a boolean algebra by explicitly constructing the required operations on finitely complemented covers (Corollary 3.2.36) and then show that $\mathcal{F}(X)$ is isomorphic to $\mathcal{N}(X)$ (Proposition 3.2.37).

In the rest of this section, we suppose that the poset X is finitely complemented.

Definition 3.2.32. Given a poset X, and two cubes $I = (x, y)$ and $J = (u, v)$ of X. We define their *intersection*, written $I \cap J$ as the following cube, when defined (i.e. when $x \vee x' \leq y \wedge y'$):

$$(x, y) \cap (u, v) = (x \vee u, y \wedge v)$$

Definition 3.2.33. We define the following operations on covers R, S in $\mathcal{R}_{\mathcal{F}}(X)$:

- 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: $\overline{R}^N = \bigcap_{(x,y)\in R}{}_N \left(\{(\bot, y') \mid y' \in \max(\{x\}^{\downarrow})\} \cup \{(x', \top) \mid x' \in \min(\{y\}^{\uparrow})\}\right)$

Lemma 3.2.34. *The above operations are well-defined on $\mathcal{R}_{\mathcal{F}}(X)$.*

Proof. Given $(X, \leq)$, a finitely complemented bounded lattice.

- $\cup_N$. Trivial.
- $\cap_N$. Given $R, S \in \mathcal{R}_{\mathcal{F}}(X)$, $(x, y) \in R \cap_N S$. By definition, $(x, y) = (x_R \vee x_S, y_R \wedge y_S)$ such that $(x_R, y_R), (x_S, y_S) \in R \times S$ both finitely complemented.

  $$\begin{aligned} \downarrow(\max x_R^{\downarrow} \cup \max x_S^{\downarrow}) &= \downarrow \max x_R^{\downarrow} \cup \downarrow \max x_S^{\downarrow} \\ &= x_R^{\downarrow} \cup x_S^{\downarrow} && \text{by Lemma 3.2.24} \\ &= \{x \in X \mid x \leq x_R \text{ or } x \leq x_s\} \\ \downarrow(\max x_R^{\downarrow} \cup \max x_S^{\downarrow}) &= (x_R \vee x_S)^{\downarrow} \end{aligned}$$

  By finitude of, $\max x_R^{\downarrow}$ and $\max x_S^{\downarrow}$, $x_R \vee x_S$ is finitely lower complemented. Dually, $y_R \wedge y_S$ is finitely upper complemented. Thus, (x, y) is finitely complemented for each cube $(x, y) \in R \cap_N S$. By definition, $R \cap_N S \in \mathcal{R}_{\mathcal{F}}(X)$

- $\overline{-}^N$. Given $R \in \mathcal{R}_{\mathcal{F}}(X)$, and $(x,y) \in R$. We have (x,y) and $(\bot,\top)$ finitely complemented. By definition of a finitely complemented poset, all $y' \in \max x^{\downarrow}$ (resp. $x' \in \min y^{\uparrow}$) are finitely upper (resp. lower) complemented. Thus, $(\bot,y')$ and $(x',\top)$ finitely complemented. Furthermore, by definition $\{(\bot,y') \mid y' \in \max x^{\downarrow}\}$ and $\{(x',\top) \mid x' \in \min y^{\uparrow}\}$ are finite. Thus, by the above proofs, as a result of a finite number of operations of union and intersection of elements of $\mathcal{R}_{\mathcal{F}}$, we have $\overline{R}^N \in \mathcal{R}_{\mathcal{F}}(X)$. □

Lemma 3.2.35. *The above operations are compatible with the corresponding ones on supports: for covers $R,S \in \mathcal{R}_{\mathcal{F}}(X)$, we have*

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

Proof. Let X a bounded lattice. Let $R,S \in \mathcal{R}_{\mathcal{F}}(X)$.

- $R \cup_N S$.

$$[R \cup_N S] = \bigcup_{I \in R \cup_N S} [I] = \bigcup_{I \in R \cup S} [I] = \bigcup_{I \in R} [I] \cup \bigcup_{J \in S} [J] = [R] \cup [S]$$

- $R \cap_N S$.

$$[R \cap_N S] = \bigcup_{(I,J) \in R \times S} [I \cap J] = \bigcup_{I \in R} \left([I] \cap \bigcup_{J \in S} [J] \right) = \bigcup_{I \in R} [I] \cap \bigcup_{J \in S} [J] = [R] \cap [S]$$

- $\overline{R}^N$. Given $R \in \mathcal{R}_{\mathcal{F}}(X)$ and $(s,t) \in R$. We have s (resp. t) finitely lower (resp. upper) complemented.

$$\begin{aligned}
[\overline{R}^N] &= [\bigcap_{(s,t) \in R}^N \overline{\{(s,t)\}}^N] \\
&= [\bigcap_{(s,t) \in R}^N (\{(\bot,x) \mid x \in \max s^{\downarrow}\} \cup \{(y,\top) \mid y \in \min t^{\uparrow}\})] \\
&= \bigcap_{(s,t) \in R} \Big(\bigcup_{x \in \max s^{\downarrow}} [\bot,x] \cup \bigcup_{y \in \min t^{\uparrow}} [(y,\top)] \Big) \\
&= \bigcap_{(s,t) \in R} \Big(\bigcup_{x \in \max s^{\downarrow}} {\downarrow} x \cup \bigcup_{y \in \min t^{\uparrow}} {\uparrow} y \Big) \\
&= \bigcap_{(s,t) \in R} ({\downarrow} \max s^{\downarrow} \cup {\uparrow} \min t^{\uparrow}) \\
&= \bigcap_{(s,t) \in R} (s^{\downarrow} \cup t^{\uparrow}) && \text{By Lemma 3.2.24} \\
&= \bigcap_{(s,t) \in R} \overline{[s,t]} \\
[\overline{R}^N] &= \overline{[R]}
\end{aligned}$$

□

The finitely complemented covers $\mathcal{R}_{\mathcal{F}}(X)$ thus form a sub-boolean algebra of $\mathfrak{P}(X)$.

Corollary 3.2.36. *The set $\mathcal{F}(X)$ is a boolean algebra.*

Proposition 3.2.37. *Given a well-ordered bounded finitely complemented lattice X, the finitely complemented supports coincide with normal ones, i.e. we have*

$$\mathcal{F}(X) = \mathcal{N}(X)$$

Proof. Given a finitely complemented well-order $(X, \leq)$, fix $Y \subseteq X$.

- $\mathcal{F}(X) \subseteq \mathcal{N}(X)$.
  Suppose $Y \in \mathcal{F}(X)$: there exists a cover $R \in \mathcal{R}_{\mathcal{F}}(X)$, such that $[R] = Y$. Let us prove that $N(\overline{Y}) = \max \overline{R}^N$. By Proposition 3.2.15 it suffices to show that for all covers S, such that $[S] \subseteq \overline{Y}$, we have $S \preceq \max \overline{R}^N$. Now suppose given such a cover S, we need to prove the following points:
  - $S \preceq \max \overline{R}^N$. By Lemma 3.2.14, $[S] \subseteq \overline{Y}$ implies $S \preceq \mathcal{C}(\overline{Y})$. Furthermore, by finitude $\overline{R}^N \preceq \max \overline{R}^N$. Thus, it suffices to show $\mathcal{C}(\overline{Y}) \preceq \overline{R}^N$.
    Given $x \in \mathcal{C}(\overline{Y})$, we have $x \in \bigcap\limits_{r \in R} \mathcal{C}(\overline{r})$. By Proposition 3.2.26, for all $r \in R$ exists $n_r \in N(\overline{r}) = \overline{\{r\}}^N$ such that $x \subseteq n_r$ i.e. $x \subseteq \bigcap\limits_{r \in R} n_r \in \overline{R}^N$. Thus, $\mathcal{C}(\overline{Y}) \preceq \overline{R}^N$.
  - $\max \overline{R}^N \preceq S \implies \max \overline{R}^N \subseteq S$. Now suppose, $\max \overline{R}^N \preceq S$. Then for all $r \in \max \overline{R}^N$ there exists $s_r \in \max \overline{R}^N$ and $r_s \in S$ such that $r \leq s_r \leq r_s$. By definition of max, $r = r_s = s_r$. Thus,

    $$\max \overline{R}^N \preceq S \implies \max \overline{R}^N \subseteq S$$

  Finally, $N(\overline{Y}) = \max \overline{R}^N$. By Lemma 3.2.35, $\max \overline{R}^N \in \mathcal{R}_{\mathcal{F}}(X)$. And by the same reasoning, $\max \overline{\overline{R}^N}^N = N(\overline{\overline{Y}}) = N(Y)$. Thus,

  $$Y \in \mathcal{F} \text{ implies } Y \in \mathcal{N}(X) \text{ and } N(Y), N(\overline{Y}) \in \mathcal{R}_{\mathcal{F}}$$

- $\mathcal{F}(X) \supseteq \mathcal{N}(X)$.
  Given $Y \in \mathcal{N}(X)$, There exists a cover $R \in \mathcal{R}(X)$ such that $[(]R) = Y$ and R in normal form. By 3.2.36, it then suffices to show that $\overline{Y} \in \mathcal{R}_{\mathcal{F}}(X)$ i.e. there exists a cover $\overline{R} \in \mathcal{R}_{\mathcal{F}}(X)$ such that $[\overline{R}] = \overline{Y}$, namely $N(\overline{Y})$. We first define an extension of $\overline{-}^N : \mathcal{R}(X) \to \mathcal{R}(X)$ to covers which are not finitely complemented, which can be seen as a sort of "best overestimation" of the normal form of the complement.

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

  We prove that for a cover in normal form R, $N(\overline{[R]}) \subseteq \overline{R}^\infty \in \mathcal{R}_{\mathcal{F}}(X)$. By the same method as for $\overline{-}^N$ in the proof of Proposition 3.2.26 and Lemma 3.2.35 for $R \in \mathcal{R}(X)$ and $r \in R$,

  $$[\overline{R}^\infty] = \overline{[R]} \text{ and } \mathcal{C}(\overline{r}) \preceq \overline{\{r\}}^\infty$$

By Remark 3.2.31, all elements of X are upper complemented i.e. all cubes $(\bot, x)$ and $(x, \top)$ with $x \in \max t^{\uparrow}$, for $t \in X$ are finitely complemented. Thus, for a finite cover R,

$$\overline{R}^{\infty} \in \mathcal{R}_{\mathcal{F}}(X)$$

Suppose given $Y \in \mathcal{N}(X)$ and $x \in \mathcal{C}(\overline{Y})$, we have $x \in \bigcap_{r \in N(Y)} \mathcal{C}(\overline{r})$. By the previous remark, for all $r \in N(Y)$ exists $n_r \in \overline{\{r\}}^{\infty}$ such that $x \subseteq n_r$ i.e. $x \subseteq \bigcap_{r \in N(Y)} n_r \in \overline{N(Y)}^{\infty}$. Thus,

$$N(\overline{Y}) \preceq \mathcal{C}(\overline{Y}) \preceq \overline{N(Y)}^{\infty}$$

By definition of the normal form, $N(\overline{Y}) \subseteq \overline{N(Y)}^{\infty}$. Thus, $N(\overline{Y}) \in \mathcal{R}_{\mathcal{F}}(X)$, finite i.e. $\overline{Y} \in \mathcal{R}_{\mathcal{F}}(X)$. By Lemma 3.2.34, the cover $N(Y)$also belongs to $\mathcal{R}_{\mathcal{F}}(X)$. i.e. $Y \in \mathcal{F}(X)$. And $N(Y)$ and $N(\overline{Y})$ are finitely complemented. □

Theorem 3.2.38. *Given a well-ordered bounded lattice X:*

The poset $\mathcal{N}(X)$ is a boolean algebra if and only if X is finitely complemented

Proof. Let X a well-ordered bounded lattice.

- If X is finitely complemented, then by Corollary 3.2.36, the set $\mathcal{F}(X)$ of finitely complemented supports form a boolean algebra, and by Proposition 3.2.37, $\mathcal{F}(X) = \mathcal{N}(X)$. Thus, $\mathcal{N}(X)$ forms a boolean algebra.
- Now suppose that X is not finitely complemented. By Remark 3.2.31, the condition (i) in the Definition 3.2.30 is always satisfied, and therefore (ii) has to be falsified. This means there exists an upper complemented element t such that there exists $s \in \min t^{\uparrow}$ which is not finitely lower complemented. Let us prove that $\mathcal{N}(X)$ is not stable by intersection, and therefore it is not a Boolean algebra. We first make two remarks.
  - By Remark 3.2.31, all elements are finitely upper complemented, namely, s and t are both finitely upper complemented.
  - Since $\bot^{\downarrow} = \emptyset$, we deduce that $\bot$ is finitely lower complemented.

  Thus, both cubes $[\bot, s]$ and $[\bot, t]$ are finitely complemented, which implies by Proposition 3.2.26, that they have a complement in normal form. Furthermore, the cover $\{(\bot, s)\}$ (resp. $\{(\bot, t)\}$) is trivially the normal form of $[\bot, s]$ (resp. $[\bot, t]$). Thus, both cubes $\overline{[\bot, t]}$ and $[\bot, s]$ belong to $\mathcal{N}(X)$. Let us show that the intersection of these two cubes of $\mathcal{N}(X)$ is not in $\mathcal{N}(X)$. Given $x \in \overline{[\bot, t]} \cap [\bot, s]$, we have $x \in t^{\uparrow}$ and $x \leq s \in \min t^{\uparrow}$. By definition of $\min t^{\uparrow}$, we deduce $x = s$. Since s is by definition not finitely lower complemented, the cube $[s, s]$ is not finitely complemented either. Thus, by Proposition 3.2.26,

  $$\overline{[\bot, t]} \cap [\bot, s] = [s, s] \notin \mathcal{N}(X)$$

  By contraposition, we deduce the desired implication.

□

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

Remark 3.2.39. Both of these boolean algebras are atomic. Indeed, they are generated by the set of supports of single cubes.

3.3 Syntactic covers of programs

3.3.1 Computing covers and complements

In the case of syntactic covers, i.e. covers 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 3.2.33. Namely,

- the union of covers is immediate to implement,
- following Proposition 3.1.32, the supremum and infimum of positions can be computed, from which we can compute the intersection of covers,
- we can compute the generators of the complement, from which we can compute the complement of covers.

Let us detail the last point.

Definition 3.3.1. Given a position p of a program P, we define, by induction on P, the following set $\inf_P(p)$ of positions of P:

$$\inf_P(\bot) = \emptyset \qquad \inf_{P;Q}(\top) = \{\top ; \top\} \qquad \inf_{P+Q}(\top) = \{\top + \bot, \bot + \top\}$$
$$\inf_A(\top) = \{\bot\} \qquad \inf_{P \parallel Q}(\top) = \{\top \parallel \top\} \qquad \inf_{P^*}(\top) = \emptyset$$

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

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

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

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

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

Proposition 3.3.2. *Given a position p of a program P, the set* $\inf_P(p)$ *is a well-defined set of positions and, we have* $\inf_P(p) = \max(p^{\downarrow})$.

Proof. Given a program P, let us prove that by induction on $x \in \mathcal{P}(P)$, that $\max(x^{\downarrow}) = \inf_P(x)$.

- $x = \bot$. Immediately,
$$\max(\bot^{\downarrow}) = \emptyset = \inf_P(\bot)$$
- $x = \top$. Let us prove $\max(\top^{\downarrow}) = \inf_P(\top)$ by induction on P.
  - If $P = Q^{\star}$. We have $\top^{\downarrow} = \{\bot\} \cup \{q^n \mid q \in \mathcal{P}(Q), n \in \mathbb{N}\}$ such that for all $q^n \in \top^{\downarrow}$, $q^n < q^{n+1} \in \top^{\downarrow}$. Thus,
$$\max(\top^{\downarrow}) = \emptyset = \inf_P(\top)$$
  - For all other cases, $\top^{\downarrow}$ is trivially generated by the predecessors of $\top$ (e.g. $\top \parallel \top$, $\varnothing + \top$ and $\top + \varnothing$ ...).

  Thus, $\max(\top^{\downarrow}) = \inf_P(\top)$.
- $P \vDash \bot ; \bot$, or $P \vDash \bot^0$ or $P \vDash \bot \parallel \bot$ or $P \vDash \bot + \bot$. By the inferences rules
$$\max(x^{\downarrow}) = \{\bot\} = \inf_P(x)$$
- $Q ; R \vDash q ; \bot$, with $q \neq \bot$. We have $q ; \bot \in {\downarrow}\{\top ; v \mid R \vDash v\}$ i.e. $x^{\downarrow} \cap \{\top ; v \mid R \vDash v\} = \emptyset$. Thus,
$$\begin{aligned} \max x^{\downarrow} &= \max\{u ; \bot \mid u \not\geq q, Q \vDash u\} \\ &= \{P \vDash u ; \bot \mid u \in \max q^{\downarrow}\} \\ &= \{P \vDash u ; \bot \mid u \in \inf_Q(q)\} \\ \max x^{\downarrow} &= \inf_P(x) \end{aligned}$$
- $Q ; R \vDash \top ; r$ with $r \neq \bot$. We have $\{q ; \bot \mid Q \vDash q\} \subseteq {\downarrow}\{\top ; v \mid R \vDash v\}$. Thus, by inference rules,
$$\begin{aligned} \max x^{\downarrow} &= \max(x^{\downarrow} \cap \{\top ; v \mid R \vDash v\}) \\ &= \{\top ; v \mid v \in \max r^{\downarrow}\} \\ &= \{\top ; v \mid v \in \inf_R(r)\} \\ \max x^{\downarrow} &= \inf_P(x) \end{aligned}$$
- $Q \parallel R \vDash x = q \parallel r$ with $q \parallel r \neq \bot \parallel \bot$. By the inference rules, $q \parallel r \not\leq u \parallel v$ is equivalent to $r \not\leq v$ or $q \not\leq u$. Thus,
$$\begin{aligned} x^{\downarrow} &= \{P \vDash u \parallel v \mid q \not\leq u\} \cup \{P \vDash u \parallel v \mid r \not\leq v\} \\ x^{\downarrow} &= {\downarrow}\{P \vDash u \parallel \top \mid u \in q^{\downarrow}\} \bigcup {\downarrow}\{P \vDash \top \parallel v \mid v \in r^{\downarrow}\} \end{aligned}$$

with both sets incomparable. Thus,

$$\begin{aligned}\max x^{\downarrow} &= \{P \vDash u \| \top \mid u \in \max q^{\downarrow}\} \cup \{P \vDash \top \| v \mid v \in \max r^{\downarrow}\} \\ &= \{P \vDash u \| \top \mid u \in \inf_Q q\} \cup \{P \vDash \top \| v \mid v \in \inf_R r\} \\ \max x^{\downarrow} &= \inf_P(x)\end{aligned}$$

- $Q^* \vDash x$. We have $\mathcal{P}(P)^+ = \{q^n \mid n \in \mathbb{N}, Q \vDash q\}$
  - $Q^* \vDash \bot^n$ with $n > 0$. By the inference rule, $n \leq m$ equivalent to $x = \bot^n \leq q^m$. Thus,
    $$\max x^{\downarrow} = \max(\{q^m \mid Q \vDash q, m < n\}) = \top^{n-1} = \inf_P(x)$$
  - $Q^* \vDash y^n$ with $y \neq \bot$ and $n \neq 0$.
    By inference, $m \neq n$ implies $q^m < \bot^n \leq x$, and $m > n$ implies $q^m > x$. Thus, by inference rule
    $$\begin{aligned}\max(x^{\downarrow}) &= \max(x^{\downarrow} \cap \{q^n \mid Q \vDash q\}) \\ &= \{q^n \mid q \not\leq y\} \\ &= \{q^n \mid q \in \max y^{\downarrow}\} \\ &= \{q^n \mid q \in \inf_Q y\} \\ \max(x^{\downarrow}) &= \inf_P(x)\end{aligned}$$

- $Q{+}R \vDash x = q{+}\varnothing$. The inference rules give
  $$\begin{aligned}(q{+}\varnothing)^{\downarrow} &= \{P \vDash u{+}\varnothing \mid q \not\leq u\} \cup \{P \vDash \varnothing{+}v \mid R \vDash v\} \cup \{\bot\} \\ (q{+}\varnothing)^{\downarrow} &= {\downarrow}\{P \vDash u{+}\varnothing \mid u \in q^{\downarrow}\} \cup {\downarrow}\{\varnothing{+}\top\}\end{aligned}$$
  With $\{P \vDash u{+}\varnothing \mid u \in q^{\downarrow}\} \cap {\downarrow}\{\varnothing{+}\top\} = \{\bot\}$ and all other elements incomparable. Thus, by induction rules
  $$\begin{aligned}\max x^{\downarrow} &= \{P \vDash u{+}\varnothing \mid u \in \max q^{\downarrow}\} \cup \{\varnothing{+}\top\} \\ &= \{P \vDash u{+}\varnothing \mid u \in \inf_Q q\} \cup \{\varnothing{+}\top\} \\ \max x^{\downarrow} &= \inf_P(x)\end{aligned}$$

- $Q{+}R \vDash x = \varnothing{+}r$. Similarly, $\max(x^{\downarrow}) = \inf_P(x)$.

Thus, for all P and all $x \in \mathcal{P}(P)$, $\max(x^{\downarrow}) = \inf_P(x)$. □

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 program satisfy the conditions of previous section:

Proposition 3.3.3. *Given a program P, its poset of positions $\mathcal{P}(P)$ is a finitely complemented well-ordered lattice.*

Proof. Essentially, the main position which is not finitely lower complemented is $\top$ in programs on the form $Q^\star$. For each program P, we thus define a predicate $\vdash$ on its positions such that $P \vdash p$ if and only if p not finitely lower complemented, by taking the closure under context of the above position:

$$\frac{Q \vdash q}{Q;R \vdash q;\bot} \qquad \frac{Q \vdash q}{Q+R \vdash q+\varnothing} \qquad \frac{Q \vdash q}{Q \parallel R \vdash q \parallel r} \qquad \frac{}{Q^\star \vdash \top}$$

$$\frac{R \vdash r}{Q;R \vdash \top;r} \qquad \frac{R \vdash r}{Q+R \vdash \varnothing+r} \qquad \frac{R \vdash r}{Q \parallel R \vdash q \parallel r} \qquad \frac{Q \vdash q \quad n \in \mathbb{N}}{Q^\star \vdash q^n}$$

We can remark that for each $p \in \mathcal{P}(P)$ and $x \in \sup_P(p)$ we have $P \nvdash x$. As we already proved that $\sup_P(p) = \min(p^\uparrow)$, to prove the finitely complemented nature of $\mathcal{P}(P)$ we prove for all $p \in \mathcal{P}(P)$, $p \vdash P$ if and only if p not finitely lower complemented, by induction on p.

- $p = \bot$. This implies $p^\downarrow = \emptyset$. Trivially $p \nvdash$ and $p \in \mathcal{P}(P)$
- $p = \top$. We remark for each $t \in \mathcal{P}(P), t \leq \top$. This implies $p^\downarrow = \downarrow \top \setminus \{\top\}$
  - $P = A$. We have $p^\downarrow = \{\bot\} = \downarrow\{\bot\}$. Thus, $P \nvdash p$ and p finitely lower complemented by Lemma 3.2.24.
  - $P = Q;R$, $P = Q+R$ and $P = Q \parallel R$. Similarly, there exists a maximum for $\downarrow p$ (namely $\top;\top$, $\top+\varnothing$ and $\varnothing+\top$, $\top \parallel \top$).
  - $P = Q^\star$. Given $x \in \top^\downarrow \setminus \bot = \{q^n \mid n \in \mathbb{N}, Q \vDash q\}$, such that $x = y^n$. By definition, $x < y^{n+1} \in \top^\downarrow$. Thus, $\max \top^\downarrow = \emptyset$. And $Q^\star \vdash \top$ by definition

We now prove the induction step.

- $p = q \parallel r$. Using the inference rule,

$$\begin{aligned}(q \parallel r)^\downarrow &= \{y \parallel z \mid y \not\geq q \text{ or } z \not\geq r\} \cup \{\bot\} \\ &= \downarrow(\{y \parallel \top \mid y \not\geq q\} \cup \{\top \parallel z \mid z \not\geq q\}) \\ &= (q \parallel \top)^\downarrow \cup (\top \parallel r)^\downarrow\end{aligned}$$

  As, $\{y \parallel \top \mid y \not\geq q\}$ and $\{\top \parallel z \mid z \not\geq q\}$ disjoint, $p^\downarrow$ is finitely lower generated if and only if both $(q \parallel \top)^\downarrow$ and $(\top \parallel r)^\downarrow$ are finitely lower generated. By induction hypothesis this is equivalent to $Q \nvdash q$ and $R \nvdash r$. And by the inference rules of $\vdash$, this is equivalent to $Q \parallel R \nvdash p$.

- The other cases are treated with a similar argument (with a single variable) p finitely lower generated equivalent to $P \nvdash p$.

□

The operations do not in general preserve the property of being normal for regions, but Theorem 3.2.38 and Proposition 3.3.3 however ensure that the property of being normalizable is preserved, and the normal form of a cover can be computed as follows.

Proposition 3.3.4. *With the implementation of operations described above, the normal form of a cover R is $N(R) = \max(\overline{\overline{R}})$, i.e. it can be obtained by computing twice the complement of R and only keeping cubes which are maximal w.r.t. inclusion.*

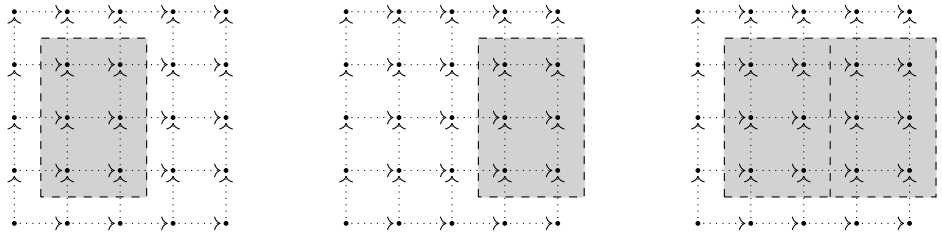
Proof. It can be observed that the definition of the complement is such that it contains all the maximal cubes. □

Proposition 3.3.5. *When the normal cover $N(R)$ is defined, it is equal to the set of all maximal cubes of the support, i.e.*

$$N(R) = \mathcal{C}^{\max}(R) = \{c \in \mathcal{C}(R) \mid \forall c' \in \mathcal{C}(R), c \not\subset c'\}$$

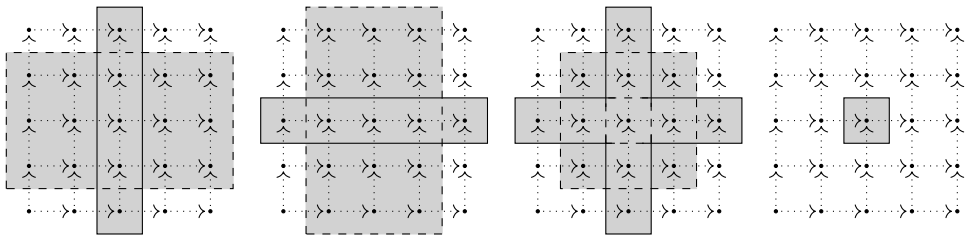
Proof. By definition $\mathcal{C}^{\max}(R)$ is maximal for the order Definition 3.2.11 □

Example 3.3.6. 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 cover R and S below, the cover $R \cup_N S$ is not normal (the normal form is the single cube covering the whole region):



R S $R \cup_N$

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



R S $R \cap_N S$ I

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

3.3.2 Implementation

We will now explain how to implement the Algorithm 1.4.33 on our syntactic model. First we will have to tweak the cubical partition from Definition 1.4.18 in order to go beyond the scope of simple programs.

3.3.2.1 Deadlock algorithm and cubical partition

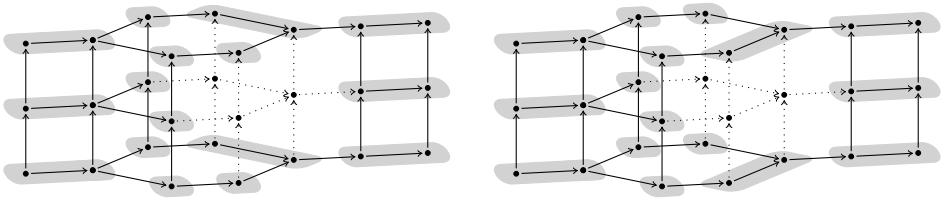
For this algorithm, we will restrict ourselves to programs of the form $P = P_1 \| \cdots \| P_n$, where P_i is a sequential process (it may contain loops and conditional branchings, but no parallel composition operator).

To implement the algorithm, we need a way to compute a cubical partition, compatible with the normal cover of our allowed region.

Remark 3.3.7. With our definition of cubes (even it the case of non-looping programs) it doesn't make sense to talk about the *coarsest partition* compatible, as it does not necessarily exist. Indeed, let us look at the program:

$$((\mathtt{P}_a\mathtt{+P}_a)\mathtt{;V}_a)\|(\mathtt{P}_a\mathtt{;V}_a)$$

There are two coarsest cubical partition (compatible with the normal cover) given below:



And fundamentally, there is no "right" choice of what should be the "good" partition, between any of these partitions.

Thus, if we were to use Definition 1.4.18 directly, we would not get a partition, as seen in Example 3.3.13 below. To get back a partition, we will iterate Γ^m until we reach a fixpoint.

We recall the operators used in Definition 1.4.18.

Definition 3.3.8. Given a cover R on a sequential program P, we define the following partition operator

$$\Gamma_1^m(R) = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} \mathcal{C}^{\max}(Z_{U,V})$$

with $Z_{U,V} = \bigcap\limits_{u \in U} [u] \bigcap\limits_{v \in V} [v]^{\mathrm{c}}$

Remark 3.3.9. If we compute the intersection of covers $\bigcap\limits_{u \in U} u \bigcap\limits_{v \in V} v^{\mathrm{c}}$, we get a cover on $Z_{U,V}$, and we can compute $\mathcal{C}^{\max}(Z_{U,V})$ using Proposition 3.3.4 using cubes

Definition 3.3.10. Given a cover R on the parallel composition of n sequential programs $P = P_1 \| \cdots \| P_n$, we define the *pre-generic cubical partition*

$$\Gamma_n^m(R) = \prod_{i=1}^{n} \Gamma_1^m\big(\{c_i \mid \exists (r_k)_{1 \leq k \leq n} \in R, r_i = c_i\}\big)$$

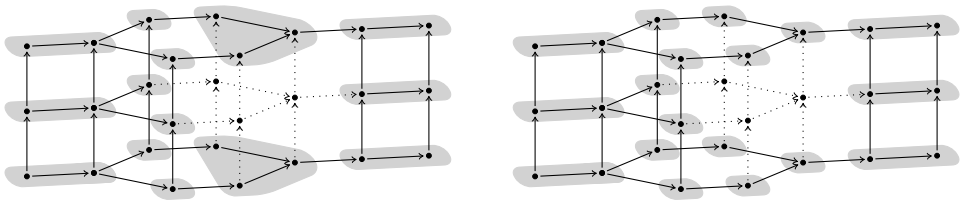
Remark 3.3.11. When $n = 1$, $\Gamma_n^m = \Gamma_1^m$, so we will simply write Γ^m when context makes it clear.

Now we define the *generic cubical partition* associated to a cover R as the fixpoint of the iterated composition by Γ^m

Definition 3.3.12. Given a cover R on a program P, we define then *generic cubical partition* $\Gamma^*(R)$ associated to a cover R as:

$$\Gamma^*(R) = \lim_{n\to\infty} \underbrace{\Gamma^m \circ \cdots \circ \Gamma^m}_{n \text{ times}}(R)$$

Example 3.3.13. If we reconsider the example from Remark 3.3.7, we give the iterated application of Γ^m on the maximal cubes of the allowed cover below.



$\Gamma^m(R)$ $\qquad$ $\Gamma^*(R) = \Gamma^m \circ \Gamma^m(R)$

In this case, $\Gamma^*(R) = \Gamma^m \circ \Gamma^m(R)$. We will prove later on, that this is the case for any maximal cover R of a program.

Remark 3.3.14. When Γ^* is the coarsest partition, it is obtained in a single iteration (which is the case for simple programs).

For sequential programs, the generic cubical partition of a maximal cover is actually the coarsest cubical partition i.e. we simply need to apply Γ^m once to the cover to recover our partition. In itself, this is not very interesting, as we interested in parallel composition, but it is a central argument to prove that Γ^* can be computed in a finite number of steps (Proposition 3.3.22).

Lemma 3.3.15. *Let R a maximal cover on a program $\mathcal{P}(P+Q)$. Then*

- *Given $(\bot, p'+\varnothing) \in R$, then $(\bot, p') \in \mathcal{C}^{\max}(\{x \mid x+\varnothing \in [R]\})$*
- *Given $(p+\varnothing, p'+\varnothing) \in R$, then $(p, p') \in \mathcal{C}^{\max}(\{x \mid x+\varnothing \in [R]\})$*

Similar properties are easily deduced for the other cases.

Proof. Given a cover R on $\mathcal{P}(P+Q)$

- Given $(p+\varnothing, p'+\varnothing) \in R$. Let us suppose $(p,p') \subseteq (q,q') \in \mathcal{C}^{\max}(\{x \mid x+\varnothing \in [R]\})$. Then, by definition $(q+\varnothing, q'+\varnothing) \subseteq [R]$, thus by maximality, $p = q$ and $p' = q'$. Thus, (p,p') is maximal.
- Given $(\bot, p'+\varnothing) \in R$. Let us suppose $(\bot,p') \subseteq (q,q') \in \mathcal{C}^{\max}(\{x \mid x+\varnothing \in [R]\})$. Then, by definition $q = \bot$ and $(\bot+\varnothing, q'+\varnothing) \subseteq [R]$. By induction rules, $(\bot, q'+\varnothing) \subseteq [R]$. Thus, by maximality, $p' = q'$. Thus, $(\bot, p')$ is maximal.

□

Lemma 3.3.16. *Let R a maximal cover on a program $\mathcal{P}(P;Q)$. Then*

- *Given $(p;\bot,q;\bot) \in R$, then $(p,q) \in \mathcal{C}^{\max}(\{x \mid x;\bot \in R\})$*
- *Given $(\bot,p;\bot) \in R$, then $(\bot,q) \in \mathcal{C}^{\max}(\{x \mid x;\bot \in R\})$*
- *Given $(p;\bot,\top;q) \in R$, then*

$$(p,\top) \in \mathcal{C}^{\max}(\{x \mid x;\bot \in R\}) \text{ and } (\bot,q) \in \mathcal{C}^{\max}(\{x \mid \top;x \in R\})$$

Similar properties are easily deduced for the other cases.

Proof. Given a maximal cover R on $P;Q$.

- Let us suppose $(p,q) \subseteq (p',q') \in \mathcal{C}^{\max}(\{x \mid x;\bot \in [R]\})$. Then, by definition $(p;\bot,q';\bot) \subseteq [R]$, thus by maximality, $p=q$ and $p'=q'$. Thus, (p,q) is maximal.
- Similarly, $(\bot,q) \subseteq (p',q') \in \mathcal{C}^{\max}(\{x \mid x;\bot \in [R]\})$, implies $(\bot,q';\bot) \subseteq [R]$, and the maximality of $(\bot,q;\bot)$ implies $q=q'$ and thus, $(\bot,q)=(p',q')$ maximal.
- Let us suppose $(p,\top) \subseteq (p',\top) \in \mathcal{C}^{\max}(\{x \mid x;\bot \in [R]\})$. Then, by definition $(p';\bot,\top;q) \subseteq [R]$, thus by maximality, $p=q$ and $p'=q'$. Thus, $(p,\top)$ is maximal. Similarly, $(\bot,q)$ is maximal.

□

We'll say that two cubes are *disconnected* when there exists no path from an element of the support of the first cube to the second one, that only crosses elements of their respective support. This is important when considering the maximal cube of the partitions, as two disconnected cubes/supports can never be joined together to form a bigger cube.

Definition 3.3.17. Let $R=(R_i)_{i\in I}$ be a cubical partition of the poset X. We define the partial order $\lhd^*$ on elements of R as the reflexive transitive closure of the following relation: $R_i \lhd R_j$ if and only if for all $x \in [R_i]$ there exists $y \in [R_j]$ and a path of $[R_i]\bigcup[R_j]$ from x to y

Definition 3.3.18. Given a program P, two cubes $i,j \in \mathcal{C}(P)$, and $\lhd$ defined in Definition 3.3.17, we say that i and j are *disconnected* when $\{i\} \not\lhd \{j\}$ and $\{j\} \not\lhd \{i\}$.

Proposition 3.3.19. *Given a maximal cover R on a sequential program P.*

$$\Gamma^*(R) = \Gamma^m(R)$$

Proof. To prove this, we only need to prove that the cubes of $\Gamma^m(R)$ are all disjoint i.e. for any partition $U \amalg V = R$, $U \neq \emptyset$, the cubes of $\mathcal{C}^{\max}(Z_{U,V})$ are disjoint.

If $(\bot,\top) \in R$. Then, by maximality $R=(\bot,\top)$. Thus, $\Gamma^m(R)=R$ and $\Gamma^*(R)=\Gamma^m(R)$. For the rest of the proof we can then suppose

$$(\bot,\top) \notin R$$

By Definition 3.2.32, as $U \neq \emptyset$, we know that there exists a cube c of P such that $\bigcap_{u \in U} u = c$. Furthermore, $(\bot, \top) \notin U$ implies that $c \neq (\bot, \top)$. Now we will prove by induction on P that for any $v \in R$ and any $c \subseteq u \in U$, $[c] \cap [v]^{\mathrm{c}} = \coprod_{i \in I} [c_i]$ where all c_i are maximal and pairwise disconnected.

First let us remark that if c or v is a singleton, the property is immediately verified. Indeed,

- If $[c] = \{x\}$ then$[c] \cap [v]^{\mathrm{c}} = \{x\}$ or $\emptyset$
- If $[v] = \{x\}$ then
  - $\{x\} \subset [c]$ contradicts the maximality of v
  - $\{x\} = [c]$ implies $[c] \cap [v]^{\mathrm{c}} = \emptyset$
  - $\{x\} \cap [c] = \emptyset$ implies $[c] \cap [v]^{\mathrm{c}} = c$

Then

- $P = \alpha$. OK.
- $P = P;Q$. Let us remark that the cubes of $P;Q$ that are not $(\bot, \top)$ or singletons are all are in one of the following sets:

$$
\begin{array}{ll}
C_1 = \{(\bot, q; \bot)\} & C_7 = \{(\top; p, \top)\} \\
C_2 = \{(\bot, \top; q) \mid q > \bot\} & C_6 = \{(p; \bot, \top) \mid p < \top\} \\
C_3 = \{(p; \bot, q; \bot) \mid p < q\} & C_5 = \{(\top; p, \top; q) \mid p < q\} \\
C_4 = \{(p; \bot, \top; q)\} &
\end{array}
$$

  - If $c \in C_1$ i.e. $c = (\bot, q; \bot)$
    - $v \in C_1$ i.e. $v = (\bot, q'; \bot)$. Then

$$
\begin{aligned}
[c] \cap [v]^{\mathrm{c}} &= [\bot, q; \bot] \cap [\bot, q'; \bot]^{\mathrm{c}} \\
[c] \cap [v]^{\mathrm{c}} &= [\bot; \bot, q; \bot] \cap [\bot, q'; \bot]^{\mathrm{c}}
\end{aligned}
$$

      Thus we have reduced to the case $c \in C_3$, $v \in C_1$.
    - $v \in C_2$. Then $c \cap v^{\mathrm{c}} = \emptyset$.
    - $v \in C_3$. Then

$$
\begin{aligned}
[c] \cap [v]^{\mathrm{c}} &= \{\bot\} \cup \left([\varnothing + \bot, \varnothing + q] \cap [\varnothing + p', \varnothing + q']^{\mathrm{c}}\right) \\
&= \{\bot\} \cup \left(\{x + \varnothing \mid x \in [\bot, q] \cap [p', q']\}\right)
\end{aligned}
$$

      By the case $c \in C_3$ and $v \in C_3$, where we can apply the induction hypothesis by Lemma 3.3.16,

$$
\{x + \varnothing \mid x \in [\bot, q] \cap [p', q']\} = \coprod_{i \in I} [c_i]
$$

      With all c_i maximal and disconnected.

· If there are no $i \in I$ such that $c_i = (\bot; \bot, q_i)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot, \bot] \coprod_{i \in I} [c_i]$$

And all cubes are disconnected and maximal.

· Otherwise, there is at most a single $j \in I$ such that $c_j = (\bot; \bot, q_j)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot, q_j] \coprod_{\substack{i \in I \\ i \neq j}} [c_i]$$

and all cubes are disconnected and maximal.

∗ $v \in C_4 \cup C_6$ i.e. $v = (p'; \bot, q')$, $q' \geq \top; \bot$ Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot, q; \bot] \cap [p'; \bot, q']^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [\bot, q; \bot] \cap [p'; \bot, \top; \bot]^{\mathrm{c}}$$

Thus we have reduced to the case, $v \in C_3$, where we can apply the induction hypothesis by Lemma 3.3.16.

∗ $v \in C_5 \cup C_7$. Then $c \cap v^{\mathrm{c}} = c$.

– If $c \in C_2$ i.e. $c = (\bot, \top; q)$.

∗ $v \in C_1 \cup C_3$. Then $v \subset c$, which contradicts maximality of v.

∗ $v \in C_2$ i.e. $v = (\bot, \top; q')$ Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot, \top; q] \cap [\bot, \top; q']^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [\bot; \bot, \top; q] \cap [\bot, \top; q']^{\mathrm{c}}$$

Thus we have reduced to the case $c \in C_4$, $v \in C_1$.

∗ $v \in C_4$i.e. $v = (p'; \bot, \top; q')$. Then

$$[c] \cup [v] \subseteq [\bot, \top; \bot] \cup [\top; \bot, \top; q'] = [\bot, \top; q']$$

Such that $v \subset (\bot, \top; q') \in \mathcal{C}([R])$, which contradicts its maximality.

∗ $v \in C_5$ i.e. $v = (\top; p', \top; q')$. Then,

· $p' = \bot$ implies

$$[c] \cup [v] \subseteq [\bot, \top; \bot] \cup [\top; \bot, \top; q'] = [\bot, \top; q']$$

Such that $v \subset (\bot, \top; q') \in \mathcal{C}([R])$, which contradicts its maximality.

· $p' > \bot$.

$$[c] \cap [v]^{\mathrm{c}} = [\bot, \top; q] \cap [\top; p', \top; q']^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [\bot, \top; \bot] \bigcup [\top; \bot, \top; q] \cap [\top; p', \top; q']^{\mathrm{c}}$$

Then, by the case $c \in C_5, v \in c_5$, we have that there exists a family of maximal disconnected cubes $(c_i)_{i \in I}$ such that

$$[\top; \bot, \top; q] \cap [\top; p', \top; q']^{\mathrm{c}} = \coprod_{i \in I} c_i$$

Furthermore, $\top;\bot \in [\top;\bot,\top;q] \cap [\top;p',\top;q']^{\mathrm{c}}$ implies that there exists a single $j \in I$ such that $c_j = (\top;\bot,q_j)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot,q_j] \coprod_{\substack{i\in I\\ i\neq j}} [c_i]$$

and all cubes are disconnected and maximal.

* $v \in C_6$. Then $[\bot,\top] = [c] \cup [v] \subseteq [R]$ implies $(\bot,\top) \in R$ which contradicts the maximality of v.
* $v \in C_7$ i.e. $v = (\top;p',\top)$. Then,

$$[c] \cap [v]^{\mathrm{c}} = [\bot,\top;q] \cap [\top;p',\top]^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [\bot,\top;q] \cap [\top;p',\top;\top]^{\mathrm{c}}$$

Thus we have reduced to the case, $v \in C_5$ or the singleton, where we can apply the induction hypothesis by Lemma 3.3.16.

– If $c \in C_3$, i.e. $c = p;\bot, q;\bot$

* $v \in C_1$. i.e. $v = (\bot,q';\bot)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,q;\bot] \cap [\bot,q';\bot]^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [\bot;\bot,q;\bot] \cap [\bot;\bot,q';\bot]^{\mathrm{c}}$$

Thus we have reduced to the case, $v \in C_3$, where we can apply the induction hypothesis by Lemma 3.3.16.

* $v \in C_2 \cup C_7$. Then $c \cap v^{\mathrm{c}} = c$.
* $v \in C_3$ i.e. $v = (p';\bot,q';\bot)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,q;\bot] \cap [p';\bot,q';\bot]^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = \{x;\bot \mid x \in [p,q] \cap [p',q']\}$$

By induction hypothesis (which can be applied, by Lemma 3.3.15), $[p,q] \cap [p',q'] = \coprod_{i\in I} [p_i,q_i]$ with all $((p_i,q_i))_{i\in I}$ disconnected and maximal. Thus, by construction,

$$\{x;\bot \mid x \in [p,q] \cap [p',q']\} = \coprod_{i\in I} [p_i;\bot,q_i;\bot]$$

with all $((p_i;\bot,q_i;\bot))_{i\in I}$ disconnected and maximal.

* $v \in C_4 \cup C_6$, i.e. $v = (p';\bot,q')$, $q' \geq \top;\bot$ Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,q;\bot] \cap [p';\bot,q']^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,q;\bot] \cap [p';\bot,\top;\bot]^{\mathrm{c}}$$

Thus we have reduced to the case, $v \in C_3$, where we can apply the induction hypothesis by Lemma 3.3.16.

* $v \in C_5$ i.e. $v = \top;p',\top;q'$

- $q, p' \neq \top, \bot$ implies $c \cap v^{\mathrm{c}} = c$
- Otherwise $p < q = \top$ implies $v \subset p;\bot,\top;q'$ which contradicts its maximality.

– If $c \in C_4$.

* $v \in C_1$ i.e. $v = (\bot, q';\bot)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;q] \cap [\bot, q';\bot]^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;q] \cap [\bot;\bot, q';\bot]^{\mathrm{c}}$$

Thus we have reduced to the case $c \in C_4$, $v \in C_3$, where we can apply the induction hypothesis by Lemma 3.3.16.

* $v \in C_2$ i.e. $v = (\bot, \top;q')$, $q' > \bot$ Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;q] \cap [\bot,\top;q']^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [\top;\bot,\top;q] \cap [\bot,\top;q']^{\mathrm{c}}$$

Thus we have reduced to the case $c \in C_5, v \in C_2$ or to the case of $[c]$ singleton.

* $v \in C_3$ i.e. $v = (p';\bot, q';\bot)$. Then $q' = \top$ is a special case of the case of $v \in C_4$. So we can suppose $q' < \top$. Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;q] \cap [p';\bot, q';\bot]^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;\bot] \cap [p';\bot, q';\bot]^{\mathrm{c}} \coprod [\top;\bot,\top;q]$$

Then, by the case $c \in C_2, v \in c_3$, where we can apply the induction hypothesis by Lemma 3.3.16, we have that there exists a family of maximal disconnected cubes $(c_i)_{i\in I}$ such that

$$[p;\bot,\top;\bot] \cap [p';\bot, q';\bot]^{\mathrm{c}} = \coprod_{i\in I} c_i$$

Furthermore, $\top;\bot \in [p;\bot,\top;\bot] \cap [p';\bot, q';\bot]^{\mathrm{c}}$ implies that there exists a single $j \in I$ such that $c_j = (p_j, \top + \bot)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [p_j, \top + q] \coprod_{\substack{i\in I \\ i\neq j}} [c_i]$$

and all cubes are disconnected and maximal.

* $v \in C_4$ i.e. $v = (p';\bot,\top;q')$. Then

$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;q] \cap [p';\bot,\top;q']^{\mathrm{c}}$$
$$[c] \cap [v]^{\mathrm{c}} = [p;\bot,\top;\bot] \cap [p';\bot,\top;\bot]^{\mathrm{c}} \coprod [\top;\bot,\top;q] \cap [\top;\bot,\top;q']^{\mathrm{c}}$$

With the cubes of $[p;\bot,\top;\bot]\cap[p';\bot,\top;\bot]^{\mathrm{c}}$ and $[\top;\bot,\top;q]\cap[\top;\bot,\top;q']^{\mathrm{c}}$ disconnected, as $\top;\bot$ in both complements. Thus, we reduced to the case $c \in C_3, V \in C_3$ and $c \in C_5, v \in C_5$, where we can apply the induction hypothesis by Lemma 3.3.16.

  - $v \in C_5$. Dual to $v \in C_3$.
  - $v \in C_6$. Dual to $v \in C_2$.
  - $v \in C_7$. Dual to $v \in C_1$.
- If $c \in C_5$. Dual to the case $c \in C_3$.
- If $c \in C_6$. Dual to the case $c \in C_2$.
- If $c \in C_7$. Dual to the case $c \in C_1$.

- $P = P+Q$. Let us start by remarking that the cubes of $P+Q$ that are not $(\bot, \top)$ all are in one of the following sets:

$$\begin{array}{ll} C_1 = \{(\bot, \varnothing+q)\} & C_6 = \{(\varnothing+p, \top)\} \\ C_2 = \{(\bot, q+\varnothing)\} & C_5 = \{(p+\varnothing, \top)\} \\ C_3 = \{(p+\varnothing, q+\varnothing) \mid p \leq q\} & C_4 = \{(\varnothing+p, \varnothing+q) \mid p \leq q\} \end{array}$$

  - If $c \in C_3$ i.e. $c = (p+\varnothing, q+\varnothing)$.
    - $v \in C_1 \cup C_4 \cup C_6$. Then $[c] \cap [v]^{\mathrm{c}} = [c]$.
    - $v \in C_2$ i.e. $v = (\bot, q'+\varnothing)$. Then

$$\begin{aligned} [c] \cap [v]^{\mathrm{c}} &= [p+\varnothing, q+\varnothing] \cap [\bot, q'+\varnothing]^{\mathrm{c}} \\ [c] \cap [v]^{\mathrm{c}} &= [p+\varnothing, q+\varnothing] \cap [\bot+\varnothing, q'+\varnothing]^{\mathrm{c}} \end{aligned}$$

      Thus we have reduced this case to a special case of $c \in C_3$ and $v \in C_3$, where the induction hypothesis can be applied by Lemma 3.3.15
    - $v \in C_3$ i.e. $c = (p'+\varnothing, q'+\varnothing)$. Then

$$\begin{aligned} [c] \cap [v]^{\mathrm{c}} &= [p+\varnothing, q+\varnothing] \cap [p'+\varnothing, q'+\varnothing]^{\mathrm{c}} \\ [c] \cap [v]^{\mathrm{c}} &= \{x+\varnothing \mid x \in [p, q] \cap [p', q']\} \end{aligned}$$

      By induction hypothesis (which can be applied, by Lemma 3.3.15), $[p, q] \cap [p', q'] = \coprod_{i \in I} [p_i, q_i]$ with all $((p_i, q_i))_{i \in I}$ disconnected and maximal. Thus, by construction,

$$\{x+\varnothing \mid x \in [p, q] \cap [p', q']\} = \coprod_{i \in I} [p_i+\varnothing, q_i+\varnothing]$$

      with all $((p_i+\varnothing, q_i+\varnothing))_{i \in I}$ disconnected and maximal.
    - $v \in C_5$. Dual to the case $v \in C_2$.
  - If $c \in C_1$ i.e. $c = (\bot, \varnothing+q)$.
    - $v \in C_1$. i.e. $v = (\bot, \varnothing+q')$. Then

$$\begin{aligned} [c] \cap [v]^{\mathrm{c}} &= [\bot, \varnothing+q] \cap [\bot, \varnothing+q']^{\mathrm{c}} \\ [c] \cap [v]^{\mathrm{c}} &= [\varnothing+\bot, \varnothing+q] \cap [\varnothing+\bot, \varnothing+q']^{\mathrm{c}} \end{aligned}$$

      This case is a special case of $c \in C_3$ and $v \in C_3$, where we can apply the induction hypothesis thanks to the Lemma 3.3.15,

* $v \in C_2$ i.e. $v = (\bot, q' \dotplus \varnothing)$. Then $[\bot, \varnothing \dotplus q] \cap [\bot, q' \dotplus \varnothing]^{\mathrm{c}} = [\varnothing \dotplus \bot, \varnothing \dotplus q]$.
* $v \in C_3 \cup C_5$. Then, $[c] \cap [v]^{\mathrm{c}} = [c]$.
* $v \in C_4$ i.e. $v = (\varnothing \dotplus p', \varnothing \dotplus q')$. Then

$$\begin{aligned}[c] \cap [v]^{\mathrm{c}} &= \{\bot\} \cup \left([\varnothing \dotplus \bot, \varnothing \dotplus q] \cap [\varnothing \dotplus p', \varnothing \dotplus q']^{\mathrm{c}}\right) \\ &= \{\bot\} \cup \left(\{x + \varnothing \mid x \in [\bot, q] \cap [p', q']\}\right)\end{aligned}$$

By the case $c \in C_3$ and $v \in C_3$, where we can apply the induction hypothesis by Lemma 3.3.15,

$$\{x + \varnothing \mid x \in [\bot, q] \cap [p', q']\} = \coprod_{i \in I} [c_i]$$

With all c_i maximal and disconnected.

  · If there are no $i \in I$ such that $c_i = (\varnothing \dotplus \bot, q_i)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot, \bot] \coprod_{i \in I} [c_i]$$

And all cubes are disconnected and maximal.

  · Otherwise, there is at most a single $j \in I$ such that $c_j = (\varnothing \dotplus \bot, q_j)$. Then

$$[c] \cap [v]^{\mathrm{c}} = [\bot, q_j] \coprod_{\substack{i \in I \\ i \neq j}} [c_i]$$

and all cubes are disconnected and maximal.

* $v \in C_6$ i.e. $v = (\varnothing \dotplus p' \top)$. Then

$$[c] \cap [v]^{\mathrm{c}} = \{\bot\} \cup \left([\bot, \varnothing \dotplus q] \cap [\varnothing \dotplus p', \varnothing \dotplus \top]^{\mathrm{c}}\right)$$

Thus we have reduced this case to the case $v \in C_4$, where we can apply the induction hypothesis thanks to the Lemma 3.3.15.

– If $c \in C_2$. Symmetric to the case $c \in C_1$
– If $c \in C_4$. Symmetric to $c \in C_3$.
– If $c \in C_5$. Dual to $c \in C_2$.
– If $c \in C_6$. Dual to $c \in C_1$

- $P = Q^*$. By assimilating Q^* to an infinite sequential composition of copies of Q and applying the case $Q;Q$ we obtain the desired result.

□

As we previously explained, for general programs of the form $P = P_1 \| \cdots \| P_n$, the generic cubical partition can be computed in only two steps of applying Γ^m.

Furthermore, as per the following Lemma 3.3.20 the computations of the iterated applications of Γ^m to a cover R can be done directly on the different projections of R along each axis. And for each of these projections, the second application of Γ^m can be done on independent subsets of cubes of $\Gamma^m(R)$ (Lemma 3.3.21)

Lemma 3.3.20. *Given a cover R on a program $P = P_1 \| \cdots \| P_n$, with P_k sequential for all $1 \leq k \leq n$. We have*

$$\Gamma^m \circ \Gamma^m(R) = \prod_{k=1}^{n} \Gamma^m \circ \Gamma^m(R_k)$$

with $R_k = \{x_k \mid \exists (r_i)_{1 \leq i \leq n} \in R, r_k = x_k\}$.

Proof. This result is obtained directly by applying the Definition 3.3.12 twice

$$\begin{aligned}
\Gamma^m \circ \Gamma^m(R) &= \Gamma^m(\prod_{k=1}^{n} \Gamma^m(R_k)) \\
&= \prod_{i=1}^{n} \Gamma^m\big(\{x_k \mid \exists (r_i)_{1 \leq i \leq n} \in \prod_{k=1}^{n} \Gamma^m(R_k), r_k = x_k\}\big) \\
&= \prod_{i=1}^{n} \Gamma^m\big(\{x_i \mid \exists r_i \in \Gamma^m(R_i), r_i = x_i\}\big) \\
\Gamma^m \circ \Gamma^m(R) &= \prod_{i=1}^{n} \Gamma^m(\Gamma^m(R_i))
\end{aligned}$$

□

As previously explained, computation of $\Gamma^m(R)$ scales greatly with the number of cubes. Furthermore, the number of cubes of $\Gamma^m(R)$ will always be greater than or equal to the number of cubes of R. Thus, when applying Γ^m multiple times, the following Lemma 3.3.21 drastically reduce computation cost by instead allowing us to apply Γ^m to subset of $\Gamma^m(R)$.

Lemma 3.3.21. *Given a cover R on P sequential, we have*

$$\Gamma^m \circ \Gamma^m(R) = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} \Gamma^m(\mathcal{C}^{\max}(Z_{U,V}))$$

Proof. By definition

$$\Gamma^m(R) = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} \mathcal{C}^{\max}(Z_{U,V})$$

Furthermore, by definition, given $(U, V) \neq (U', V')$, we have $Z_{U,V} \cap \mathbb{Z}_{U',V'} = \emptyset$ As the $Z_{U,V}$ that are empty can be ignored, we can consider that

$$\Gamma^m(R) = \bigcup_{i \in I} \mathcal{C}^{\max}(R_i)$$

with all $R_i \neq \emptyset$ and $R_i \cap R_j = \emptyset$. Now let us suppose given a partition $X \amalg Y = \Gamma^m(R)$, $X \neq \emptyset$ and $Z_{X,Y} \neq \emptyset$. We define

$$X_i = X \cap \mathcal{C}^{\max}(R_i) \qquad\qquad Y_i = Y \cap \mathcal{C}^{\max}(R_i)$$

such that

$$Z_{X,Y} = \bigcap_{j \in I} Z_{X_j,Y_j}$$

Then by contradiction, let us prove that there exists a single $i \in I$ such that $X_i \neq \emptyset$.

- First, as $X = \bigcup_{i \in I} X_i$, and $X \neq \emptyset$, there exists $i \in I$ such that $X_i \neq \emptyset$.

- Now let us suppose by contradiction that there exists two distinct indices $i \neq j$ such that X_i and X_j are non-empty. Then let $x_i, x_j \in X_i \times X_j$. By definition,

  $$Z_{X,Y} \subseteq [x_i] \cap [x_j] \subseteq R_i \cap R_j = \emptyset$$

  Which contradicts our hypothesis $Z_{X,Y} \neq \emptyset$.

Thus, there exists a unique $i \in I$, such that $X \cap \mathcal{C}^{\max}(R_i) \neq \emptyset$. Now, let us suppose $j \in I, j \neq i$. Then as $R_i \cap R_j = \emptyset$, by definition,

$$\begin{aligned}
Z_{X_i,Y_i} \cap Z_{X_j,Y_j} &= Z_{X_i,Y_i} \cap Z_{\emptyset,\mathcal{C}^{\max}(R_j)} \\
&= Z_{X_i,Y_i} \cap \bigcap_{y \in \mathcal{C}^{\max}(R_j)} [y]^{\mathrm{c}} \\
&= Z_{X_i,Y_i} \cap (\bigcup_{y \in \mathcal{C}^{\max}(R_j)} [y])^{\mathrm{c}} \\
&= Z_{X_i,Y_i} \cap [R_j]^{\mathrm{c}} \\
Z_{X_i,Y_i} \cap Z_{X_j,Y_j} &= Z_{X_i,Y_i}
\end{aligned}$$

Then, for all $X \amalg Y$ such that $X \neq \emptyset$ and $Z_{X,Y} \neq \emptyset$, we have

$$Z_{X,Y} = \bigcap_{j \in I} Z_{X_j,Y_j} = Z_{X_i,Y_i}$$

Thus, with $Y_X = \Gamma^m(R) \setminus X$

$$\begin{aligned}
\{X \subseteq \Gamma^m(R) \mid X \neq \emptyset, Z_{X,Y_X} \neq \emptyset\} &= \bigcup_{i \in I} \{X \subseteq \mathcal{C}^{\max}(R_i) \mid X \neq \emptyset, Z_{X,Y_X} \neq \emptyset\} \\
\{X \subseteq \Gamma^m(R) \mid X \neq \emptyset, Z_{X,Y_X} \neq \emptyset\} &= \bigcup_{i \in I} \{X \subseteq \mathcal{C}^{\max}(R_i) \mid X \neq \emptyset, Z_{X,\mathcal{C}^{\max}(R_i) \setminus X} \neq \emptyset\}
\end{aligned}$$

Now we can finally prove our result

$$\begin{aligned}
\Gamma^m \circ \Gamma^m &= \bigcup_{\substack{X \amalg Y \\ X \neq \emptyset}} Z_{X,Y} \\
&= \bigcup_{i \in I} \bigcup_{\substack{X \subseteq \mathcal{C}^{\max}(R_i) \\ X \neq \emptyset}} Z_{X, \Gamma^m(R) \setminus X} \\
&= \bigcup_{i \in I} \bigcup_{\substack{X \subseteq \mathcal{C}^{\max}(R_i) \\ X \neq \emptyset}} Z_{X, \mathcal{C}^{\max}(R_i) \setminus X} \\
&= \bigcup_{i \in I} \Gamma^m(\mathcal{C}^{\max}(R_i)) \\
\Gamma^m \circ \Gamma^m(R) &= \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} \Gamma^m(\mathcal{C}^{\max}(Z_{U,V}))
\end{aligned}$$

□

Now we have everything to prove that the generic partition can indeed be obtained by applying our operator Γ^m twice, with the previous lemmas mitigating the computation costs of the second iteration.

Proposition 3.3.22. *Given a maximal cover R on a program $P = P_1 \| \cdots \| P_n$, with P_i a sequential program we have that $\Gamma^*(R) = \Gamma^m \circ \Gamma^m(R)$*

Proof. We only need to prove that all cubes of $\Gamma^m \circ \Gamma^m(R)$ are disjoint to conclude the proof. By Lemma 3.3.20 we have that

$$\Gamma^m \circ \Gamma^m(R) = \prod_{k=1}^{n} \Gamma^m \circ \Gamma^m(R_k)$$

with $R_k = \{x_k \mid \exists (r_i)_{1 \leq i \leq n} \in R, r_k = x_k\}$. By Lemma 3.3.21,

$$\Gamma^m \circ \Gamma^m(R_k) = \bigcup_{\substack{U \amalg V = R_k \\ U \neq \emptyset}} \Gamma^m(\mathcal{C}^{\max}(Z_{U,V}))$$

As all $Z_{U,V}$ are disjoint, we simply need to prove that the cubes of $\Gamma^m(\mathcal{C}^{\max}(Z_{U,V}))$ are disjoint. By definition $\mathcal{C}^{\max}(Z_{U,V})$ is maximal, we can then apply Proposition 3.3.19, which gives the desired result. □

3.3.2.2 Algorithm and implementation

With what we have proven so far in the chapter, we can finally implement Algorithm 1.4.33 for syntactic region. The algorithm is the same as before, with $\lhd$ defined on syntactic cubes in Definition 3.3.17.

Algorithm 3.3.23. Let R be the cover of the forbidden region of a program P, such that $X = \overline{R}$ is the authorized region (pruned state space) of P. We can compute

1. The normal cubical cover $N(X)$ of X as maximal cubes of the complement of X^{c}

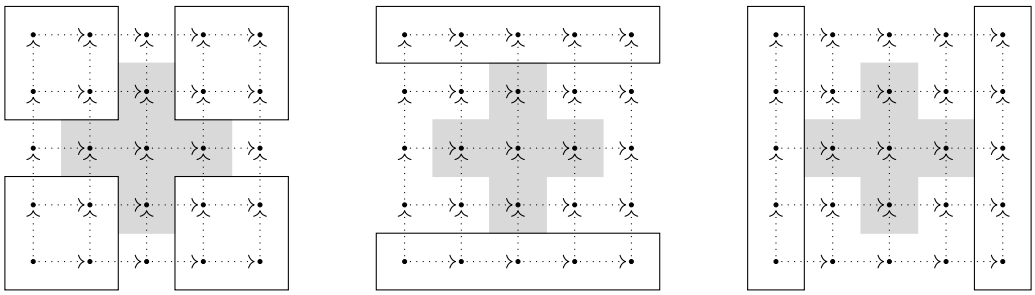
2. The generic cubical partition $\Gamma^*(N(X))$ compatible with $N(X)$

3. Deadlocks are the cubes of $\Gamma^*(N(X)) \setminus R_{max}$ that are maximal (w.r.t. $\lhd^*$). Where R_{max} is the only cube of $\Gamma^*(N(X))$ containing $\top_P$

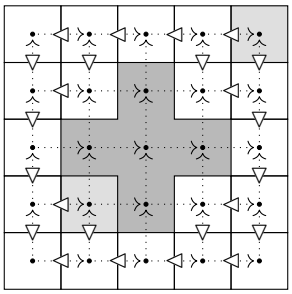
4. A cubical partition $\mathcal{U}(X)$ of the unsafe region as the downward closure (w.r.t. $\lhd$) of the deadlocks

5. A cubical partition $\mathcal{D}(X)$ of the doomed region as $\mathcal{U}(X) \setminus \mathcal{E}(X)$, where $\mathcal{E}(X)$ is the downwards closure (w.r.t. $\lhd^*$) of R_{max}

Example 3.3.24. Let us reconsider the Swiss Cross $(\mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a)\|(\mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b)$ whose maximal cubes are given below

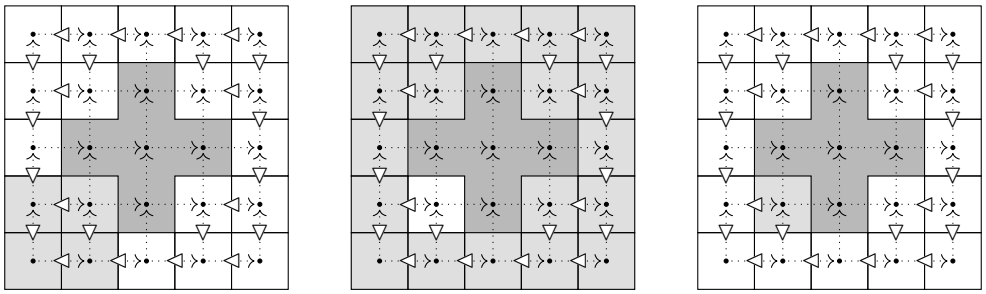


Algorithm 3.3.23 gives the following ordering of the generic cubical partition w.r.t. the maximal cover above.



In light grey in the figure above we see all cubes that do not have transition to another cube. Then we obtain, the unique deadlock of our program as the upper corner of the cube with no transitions that is not the cube in the upper corner (the end of our program).

Then, the unsafe is obtained on the left as the downwards closure of this cube. The downward closure of the maximal cube is given in the middle, and the doomed region, which correspond to the intersection of its complement with the unsafe region can be seen on the right.



The steps of the algorithm are performed the same way as in the Example 1.4.34. Now let us recapitulate how to implement them. All these were effectively implemented in [42], using the OCaml language.

Programs and positions Any inductive structure works well for implementing the definitions of Section 3.1.

Calculating the normal form. The first step, calculating the normal form of a region, requires implementing the operations in Definition 3.2.33:

- Union is union of list.
- Intersection can be implemented by Definition 3.2.32. This requires the implementation of $\vee$ and $\wedge$ of Proposition 3.1.32, which is easily implemented with our data structure.
- Complement $\dot{-}^N$ requires:
  - Intersection (which is not a problem per the point above)
  - Comparison of cubes. Though, as:

    $$(x, y) \subseteq (u, v) \iff u \leq x \text{ and } y \leq v \iff u \wedge x = u \text{ and } y \vee v = v$$

    comparison can be done with the implementation of $\vee$ and $\wedge$ of Proposition 3.1.32.
  - Calculating the generators of the lower and upper complement. This was already done in Definition 3.3.1 and Proposition 3.3.2.

Calculating the coarsest cubical partition The second step, finding the normal cubical partition requires only implementation of intersection, and the normal form of a region, whose implementation is already explained above.

Deadlocks, unsafe and doomed regions To implement these step the only additional operation to implement is comparing two cubes w.r.t. $\lhd$. This can be done in the same way as for cubical partition of the geometric semantics of a program (Proposition 1.4.32):

Proposition 3.3.25. *Given a program P, and two cubes $(p, q), (p', q') \in \mathcal{C}(P)$, such that $(p, q) \cap (p', q') = \emptyset$ we have that*

$$(p, q) \lhd (p', q') \iff \exists y' \in (p', q'), q \to y' \text{ or } \exists x \in (p, q), x \to p'$$

Proof. To prove this we will adapt [30, Proposition 7.5]. Given two disjoint cubes (p, q), (p', q') on a program P, $(p, q) \lhd (p', q')$ is by definition equivalent to: for all $u \in (p, q)$, there exists $v' \in (p', q')$ such that $u \to^* v'$ i.e.

$$\text{there exists } v, u' \in [p, q] \times [p', q'] \text{ such that } u \to^* v \to u' \to^* v' \tag{3.1}$$

Let us prove that this is equivalent to

$$\exists y' \in (p', q'), q \to y' \text{ or } \exists x \in (p, q), x \to p' \tag{3.2}$$

The indirect implication 3.2 $\implies$ Eq. (3.1) is evident. Let us prove the direct 3.1 $\implies$ Eq. (3.2) by induction on P.

First let us remark that by definition:

$$(x, v') \subseteq (p, q) \text{ and } (u', y') \subseteq (p', q')$$

Then by the fact that the cubes are disjoint we also have

$$u' \notin (p, q) \text{ and } v \notin (p', q')$$

- $P = \alpha$. OK.
- $P = P_1 ; P_2$. We will differentiate according to the value of v.
  - $v = \bot$. This implies by inference rules, $u' = \bot ; \bot$. By our first remark, this implies $p' \leq \bot ; \bot$. As $v \notin (p', q')$, we have $p' > \bot$ i.e. $p' = \bot ; \bot$. Thus taking $x = v$ concludes the case.
  - $v = v_1 ; \bot$, $a < \top$. Then $u' = u'_1 ; \bot$. Supposing that $\bot < p$ and $q' < \top$, we have $p = p_1 ; \bot$ and two cases for q':
    - $q' = q'_1 ; \bot$. As $p \leq u \leq q$ and $p' \leq v' \leq q'$ we have

      $$u = u_1 ; \bot \text{ and } v = v_1 ; \bot$$

      Then the existence of a path $u \to^* v \to u' \to v'$ is by inference rules equivalent to the existence of a path $u_1 \to^* v_1 \to u'_1 \to^* v'_1$. By induction hypothesis this is equivalent to

      $$\exists y'_1 \in (p'_1, q'_1), q_1 \to y'_1 \text{ or } \exists x_1 \in (p_1, q_1), x_1 \to p'_1$$

      By inference rules this is equivalent to

      $$\exists \in (p'_1 ; \bot, q'_1 ; \bot), q_1 ; \bot \to y'_1 ; \bot \text{ or } \exists x_1 ; \bot \in (p_1 ; \bot, q_1 ; \bot), x_1 ; \bot \to p'_1 ; \bot$$

      This concludes the case.
    - $q' = \top ; q'_2$. Remarking that $(p_1 ; \bot, q) \lhd (p', q')$ is equivalent to $(p_1 ; \bot, q) \lhd (p, \top ; \bot)$ we reduce to the previous case.

    Then, remarking that in the remaining case, $(\bot, q) \lhd (p', \top)$ is equivalent to $(\bot ; \bot, q) \lhd (p', \top ; \top)$ we conclude the case
  - $v = \top ; a$, $a < \top$. Similar to the case above.

– $v = \top ; \top$. This implies $u' = \top$ ad is dual to the case $v = \bot$.

- $P = P_1 + P_2$.
  - $v = \bot$. This implies by inference rules, $u' = \bot + \varnothing$ or $u' = \varnothing + \bot$. By our first remark, this implies $p' \leq \bot + \varnothing$ or $p' \leq \varnothing + \bot$. As $v \notin (p', q')$, we have $p' > \bot$ i.e. $p' = \bot + \varnothing$ or $p' = \varnothing + \bot$. Thus taking $x = v$ concludes the case.
  - $v = v_1 + \varnothing$, $v_1 < \top$. Then $u' = u'_1 + \varnothing$. Supposing that $\bot < p$ and $q' < \top$, we have $p = p_1 + \varnothing$ and $q' = q'_1 + \varnothing$. As $p \leq u \leq q$ and $p' \leq v' \leq q'$ we have
    $$u = u_1 + \varnothing \text{ and } v = v_1 + \varnothing$$
    By induction hypothesis this is equivalent to
    $$\exists y'_1 \in (p'_1, q'_1), q_1 \to y'_1 \text{ or } \exists x_1 \in (p_1, q_1), x_1 \to p'_1$$
    By inference rules this is equivalent to
    $$\exists \in (p'_1 + \varnothing, q'_1 + \varnothing), q_1 + \varnothing \to y'_1 + \varnothing \text{ or } \exists x_1 + \varnothing \in (p_1 + \varnothing, q_1 + \varnothing), x_1 + \varnothing \to p'_1 + \varnothing$$
    This concludes the case Then, remarking that in the remaining case, $(\bot, q) \lhd (p', \top)$ is equivalent to $(\bot + \varnothing, q) \lhd (p', \top + \varnothing)$ we conclude the case
  - $v = \varnothing + v_2$, $v_2 < \top$. Symmetric to the case above.
  - $v = \top + \varnothing$ or $v = \varnothing + \top$. Remarking that this implies $u' = \top$ it becomes dual to the case $v = \bot$

- $P = Q^\star$. By remarking that the set of positions of $Q^\star$ is the same as the set of positions of an infinite sequence of $Q; Q; Q; \cdots$, we can treat this case in much the same way as the case $P = P_1 ; P_2$.

- $P = P_1 \parallel P_2$ Remarking that $(p_1 \parallel p_2, q_1 \parallel q_2) = (p_1, q_1) \times (p_2, q_2)$, we have that
  $$(p_1 \parallel p_2, q_1 \parallel q_2) \lhd (p'_1 \parallel p'_2, q'_1 \parallel q'_2) \iff (p_1, q_1) \lhd (p'_1, q'_1) \text{ and } (p'_2, q'_2) \lhd (p'_2, q'_2)$$
  We can directly apply the induction hypothesis, which concludes the case.

□

Remark 3.3.26. One way of checking the conditions of Proposition 3.3.25 is to check that
$$[\{(p, y') \mid q \to y'\} \cap \{(p', q')\}] \cup [\{(p, q)\} \cap \{(x, q') \mid x \to p'\}] \neq \emptyset$$
The set $\{x \mid p \to x\}$ can be computed very easily using Definition 3.3.1.

Thus, we have shown that the algorithm is effectively implementable. Our implementation of this algorithm in the tool "Sparkling" can be found at [42]. This implementation was done in Ocaml to fully maximize the use of the inductive structure of our positions. In the Fig. 3.1 below, we can see the result given by the tool for the Swiss Cross from previous examples. Many more examples of code are given in the tool, such as the floating square and more complex programs such as a conservative version of the producer/consumer problem [14, Section 4.1].

For now the tool doesn't explicitly give the intermediate partition in output, but does compute it to calculate the possible deadlocks. For programs with loops, cubes whose endpoints only differ by the iteration of the loop have been regrouped, replacing the loop number by $*$.

Sparkling

Online demonstration of computations on *syntactic regions* of programs (the results you see below are computed by your browser). The source code of this program is available on the dedicated github repository.

Program

Please enter your program below:

swiss-flag

```
void main()
{
  mutex a;
  mutex b;

  {
    lock(a);
    lock(b);
    unlock(b);
    unlock(a);
  }|{
    lock(b);
    lock(a);
    unlock(a);
    unlock(b);
  };
}
```

Compute regions

Results

Finished.

Parsed program

```
?;?;(P(a);P(b);V(b);V(a)|P(b);P(a);V(a);V(b))
```

Brackets

```
b: □;□;(⊥|(⊤;_;_;_)) – □;□;(⊤|(□;□;□;⊥))
a: □;□;(⊥|(□;⊤;_;_)) – □;□;(⊤|(□;□;⊥;_))
a: □;□;((⊤;_;_;_)|⊥) – □;□;((□;□;□;⊥)|⊤)
b: □;□;((□;⊤;_;_)|⊥) – □;□;((□;□;⊥;_)|⊤)
```

Forbidden

```
□;□;((⊤;_;_;_)|(□;⊤;_;_)) – □;□;((□;□;□;⊥)|(□;□;⊥;_))
□;□;((□;⊤;_;_)|(⊤;_;_;_)) – □;□;((□;□;⊥;_)|(□;□;□;⊥))
```

Normalized:

```
□;□;((□;⊤;_;_)|(⊤;_;_;_))–□;□;((□;□;⊥;_)|(□;□;□;⊥))
□;□;((⊤;_;_;_)|(□;⊤;_;_))–□;□;((□;□;□;⊥)|(□;□;⊥;_))
```

Fundamental

```
□;□;((□;□;⊤;_)|⊥)        –□;□;(⊤|(□;⊥;_;_))
⊥                        –□;□;((□;⊥;_;_)|(□;⊥;_;_))
⊥                        –□;□;(⊤|(⊥;_;_;_))
⊥                        –□;□;((⊥;_;_;_)|⊤)
□;□;((□;□;⊤;_)|(□;□;⊤;_))–⊤
□;□;(⊥|(□;□;□;⊤))        –⊤
□;□;(⊥|(□;□;⊤;_))        –□;□;((□;⊥;_;_)|⊤)
□;□;((□;□;□;⊤)|⊥)        –⊤
```

Normalized:

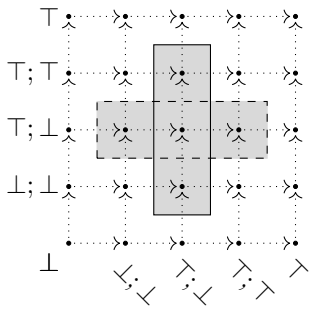
```
⊥                        –□;□;(⊤|(⊥;_;_;_))
□;□;(⊥|(□;□;⊤;_))        –□;□;((□;⊥;_;_)|⊤)
⊥                        –□;□;((⊥;_;_;_)|⊤)
⊥                        –□;□;((□;⊥;_;_)|(□;⊥;_;_))
□;□;(⊥|(□;□;□;⊤))        –⊤
□;□;((□;□;□;⊤)|⊥)        –⊤
□;□;((□;□;⊤;_)|(□;□;⊤;_))–⊤
□;□;((□;□;⊤;_)|⊥)        –□;□;(⊤|(□;⊥;_;_))
```

Deadlocks

```
□;□;((□;⊥;_;_)|(□;⊥;_;_))
```

Figure 3.1: Sparkling tool running on the Swiss Cross

Example 3.3.27. Let us look at the results of our tool Sparkling on the Swiss Cross $P = \mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a \| \mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b$ in more details. Its syntactic semantics is given below, where the notations of positions have been simplified to ease presentation. The forbidden regions in grey, is computed by looking at conflicts on the resources, giving us the two cubes below.



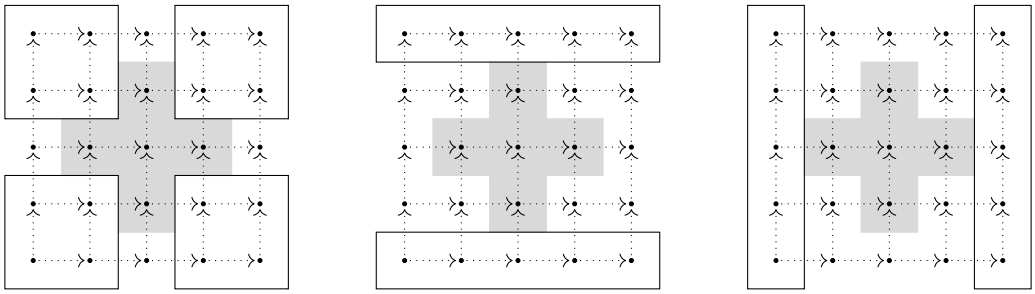
Then, the maximal cubes of the authorized region are given by Sparkling:

$\bot$	–	$((\square;\bot;\_;\_) \| (\square;\bot;\_;\_))$
$(\bot \| (\square;\square;\top;\_))$	–	$((\square;\bot;\_;\_) \| \top)$
$((\square;\square;\top;\_) \| \bot)$	–	$\square;\square;(\top \| (\square;\bot;\_;\_))$
$((\square;\square;\top;\_) \| (\square;\square;\top;\_))$	–	$\top$

These correspond to the maximal cubes in the semantics below on the left. The tool also outputs the cubes

$\bot$	–	$(\top \| (\bot;\_;\_;\_))$	$\bot$	–	$((\bot;\_;\_;\_) \| \top)$
$(\bot \| (\square;\square;\square;\top))$	–	$\top$	$((\square;\square;\square;\top) \| \bot)$	–	$\top$

with each column corresponding respectively to the middle and right cubes on the semantics below.

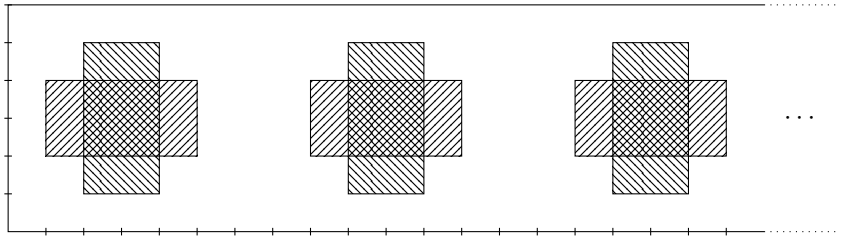


We removed the first instructions $\square;\square$ in the positions. These correspond to the declaration of mutex, which for now are in the program in input, and clutters notations without changing the structure of the program.

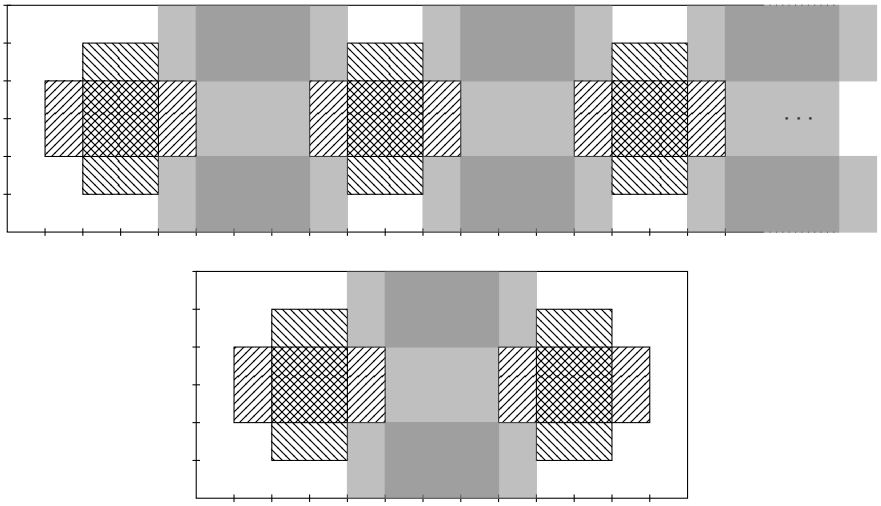
3.3.2.3 Limitations

We chose to unfold all the loops in our model in the hope of dealing with programs with loops more easily. Although it is true that this allows to implement the algorithm on some program with loops, the only actual programs we can analyse are programs where, for each loop, the positions corresponding to a loop is either totally inside or totally outside the forbidden region. But this falls apart as soon as the forbidden region does not strictly contain the loops as seen in the Example 3.3.28 below. This is due to the fact that our algorithm only works when working with finite regions, and in such programs, the (normal form of the) forbidden region is infinite.

Example 3.3.28. Let us consider the looped Swiss Cross $(\mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a)^*\|(\mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b)$, a representation of its state space is given below and extends infinitely on the right. We can see that there will be an infinite number of cubes needed to cover the forbidden region (as well as the authorized).



But looking more closely at the Example 3.3.28, we can see that the cubes from the forbidden and allowed regions are all iterations, of cubes of the loop "unfolded" only twice. For example, let us examine the cubes in between the loops in the Example 3.3.28 above. They are represented in the upper figure, and below is the 2-unfolding of the program.



These notions are not new and have already been talked about in [17], where instead of unfolding loops an infinite amount of time as we did, the authors prove that this can be done in a finite number of unfolding. Thus, in the next chapter, we will try to see if we can use these notions to apply our algorithm to programs with loops in our model.

Chapter 4
Handling programs with loops

"We dug for months, years — an eternity. And we were rewarded with madness."

– Darkest Dungeon

The deadlock detection algorithm we introduced in Section 1.4.3 and Section 3.3.2 only works for programs without loops, or only when the forbidden region does not exist within the loops. In theory, we could unfold loops an infinite number of times, using the contextual equivalence from Remark 1.1.26, but this would not be satisfactory for practical applications.

In [17] it is shown that we can unfold loops a finite number of time and still keep the main properties of our state space. For example, if there is a deadlock in our base program, it will be found in a finite unfolding of our program.

These deadlocks can be found using a different deadlock detection algorithm than the one presented in Algorithm 1.4.33 that was introduced in [19], which was adapted to work even in the presence of loops in [17] using this finite unfolding. This algorithm works by computing intersection of maximal cubes of the cover of the forbidden region. Unfortunately, the size of the forbidden region grows in $\mathcal{O}(k^n)$ where k is the number of unfolding done and n the number of parallel loops being unfolded.

Thus, keeping the number of unfolding to a minimum is crucial. For deadlock detection this is not a problem as a single unfolding suffice, but for other tasks, such as detecting the doomed region, only an upper bound is given in [17], which can get quite large.

Here, we improve upon their result, by showing that, for unsafe and doomed regions, unfolding only twice is always enough, thus resulting in much faster computations in theory. This necessitates the use of the slightly less efficient Algorithm 1.4.33 for detecting deadlocks on the unfolding. This was done to exhibit a closer connection between the two state spaces, showing that the unfolding can be done not only pointwise but can also be done by lifting the maximal covers of our regions.

We start this chapter by presenting the Algorithm 4.1.6 from [19] and the follow-up work [17] for deadlock detection in programs with loops using the unfolding.

Then we modify our syntactic model of programs from Chapter 3, folding back the loops. We also redefine a notion of syntactic cubes that works in our new semantics

(which are now pre-ordered sets instead of partial orders), and define the 2-unfolding of a semantic program.

Next, we prove that, the covers of the forbidden and authorized regions of conservative programs lifts from the base program to the unfolding and project back onto the correct regions.

Finally, we prove that applying a slightly modified version of Algorithm 3.3.23 to the unfolding of a program allows us to detect deadlocks, unsafe and doomed regions in the base program.

4.1 Finite unfolding of concurrent programs

4.1.1 Finite unfolding techniques in directed models

4.1.1.1 A second deadlock detection algorithm

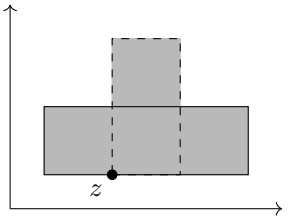
In this section, we present another algorithm for deadlock detection in loop-free concurrent programs introduced in [19], following the more recent description from [18]. This algorithm uses the forbidden region (more precisely its cubical cover) instead of the geometric semantics of the program. In most cases, the forbidden region of a concurrent program (especially program with mutexes only) is much smaller (in terms of size of the cubical cover) than the geometric semantics of the program.

The algorithm only works for space which verifies the *genericity* condition of Definition 4.1.1 below, which is always satisfied by the forbidden regions (more precisely the maximal cubical cover of the forbidden region) of a concurrent program.

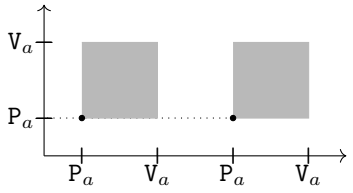
Definition 4.1.1. Let $R = \{R^1, \dots, R^m\}$ be the cubical cover of a directed space X, such that $R^j = \prod_{i \leq n} [x_i^j, y_i^j]$ for $j \leq m$. Then, R is *generic* when for each $j \neq k$, $R^j \cap R^k = \emptyset$ implies $x_i^j \neq x_i^k$ for all $i \leq n$.

Indeed, for a program with mutexes, the lower corner of cubes of the forbidden region will always correspond to the concurrent locking of a certain mutex in the corresponding projection (i.e. process) of the program. So, the fact that cubes share this coordinate implies that both process lock the same mutex. Then the upper corners of the cubes correspond to the instant where the mutex is released, meaning that both end at the same point, which would imply that one is included in the other as in Example 4.1.2. Of course, if their intersection is empty, then this is not a problem (as shown in Example 4.1.3), as it is possible of locking, unlocking and then locking again the resource.

Example 4.1.2. Let us consider the following d-space, consisting of $\vec{I} \setminus (R^1 \cup R^2)$ where $R^1 \cup R^2$ is the greyed region below, consisting of the 2 maximal dotted cubes. Then, the point z is the lower bound of an intersection of cubes, but is not a deadlock.

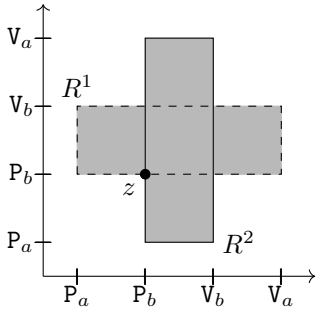


Example 4.1.3. Of course, this condition does not apply when the two cubes are disjoint, as it is possible for a program to lock and release the same mutex multiple times. For example, one could take the program $\mathtt{P}_a;\mathtt{V}_a;\mathtt{P}_a;\mathtt{V}_a \| \mathtt{P}_a;\mathtt{V}_a$ whose semantics is given below:



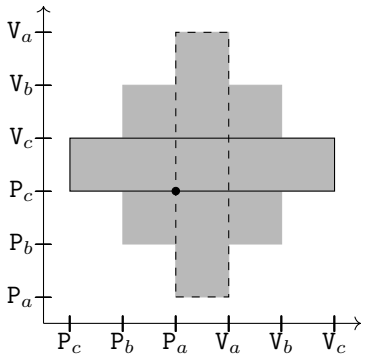
In the case of generic spaces, all deadlocks can be found as the lower bounds of the intersection $\bigcap_{i\leq n} R^i$ of n cubes R_i of the forbidden region, where n is the number of processes running in parallel in the program. Since the genericity condition above guarantees that cubes of the forbidden region do not share any coordinates if their intersection is non-empty, if we consider the lower bound of the resulting intersection, the progression of any execution from this point is blocked by exactly one cube in each direction.

Example 4.1.4. Let us consider the Swiss Cross program from Example 1.3.50. As there are two cubes R^1, R^2 in the maximal cover of the forbidden region, the only possible intersection of two cubes has the point z as lower bound. In executions from this point, the cube R^1 prevents progression along the vertical axis and R^2 along the horizontal axis:



Computing deadlocks this way might give points that are not inside the state space as in Example 4.1.5. Thus, for each of the points obtained this way, we need to check if it is at the border of the forbidden region in order to really be considered a deadlock.

Example 4.1.5. Consider the program $(\mathtt{P}_a;\mathtt{P}_b;\mathtt{P}_c;\mathtt{V}_c;\mathtt{V}_b;\mathtt{V}_a) \| (\mathtt{P}_c;\mathtt{P}_b;\mathtt{P}_a;\mathtt{V}_a;\mathtt{V}_b;\mathtt{V}_c)$, whose geometric semantics is given below. The lower bound of the intersection of the two dotted maximal cubes is inside the forbidden region, thus it should not be considered a deadlock.



This gives us the following algorithm for detecting deadlocks ([18, Algorithm 5.13]).

Algorithm 4.1.6. For X a generic d-space of dimension n, and a maximal cubical cover of its forbidden region $R = \{R^1, \dots, R^m\}$ of cardinal m, where $R^{j_k} = \prod_{i \leq n} [x_i^{j_k}, y_i^{j_k}]$. The deadlocks can be found as follows.

1. Take an intersection of n distinct cubes $R^{j_1}, \dots, R^{j_n}$ such that $\bigcap_{k \leq n} R^{j_k} \neq \emptyset$.

2. Take z such that $z_i = \max\{x_i^{j_k} \mid k \leq n\}$ (the lower bound of their intersection).

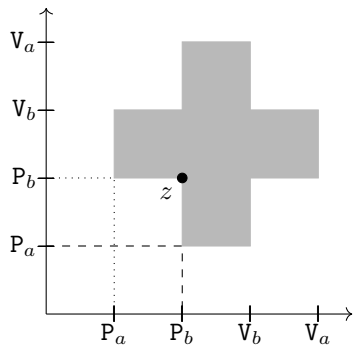
3. If for all $k \leq m$, $z \notin R^k$ then z is a "deadlock".

We refer to Example 4.1.4 for a quick illustration of this algorithm.

Remark 4.1.7. What this algorithm really detects is points without future, but these are not necessarily reachable (even when considering non-deterministic branchings). This does also affect the unsafe and doomed regions and is particularly important in programs with loops where reachability is harder to analyze, as illustrated below in Example 4.1.18. Even in programs without loops accessibility of a given point is not easily computed.

In order to find the doomed region, we can extend this and use a similar argument. Indeed, for each deadlock z, as long as a point is in the past along at least one coordinate i of the lower corner of a forbidden cube that is itself lower than z_i, there is a path going "below" that cube (otherwise, it would contradict the maximality of the cubes). And conversely, if a point is in between z_i and all other cubes' lower corner alongside i for each coordinate, it is easy to see that all paths lead to the deadlock z eventually. Once again we refer to [18, Theorem 5.11] for a more detailed proof.

Example 4.1.8. Consider the Swiss Cross from Example 4.1.4, with the singular deadlock z. It is easy to see that any point below the dashed line (representing the past of the lower corner of a cube of the forbidden region) is not doomed. Conversely, points in the past of z above both dashed line and on the right of the dotted line are doomed.



Then we can compute the doomed region with the following algorithm ([18, Algorithm 5.14]) by applying this method recursively (removing the cubes of the "doomed" regions to the state space and iterating until we reach a fixed point).

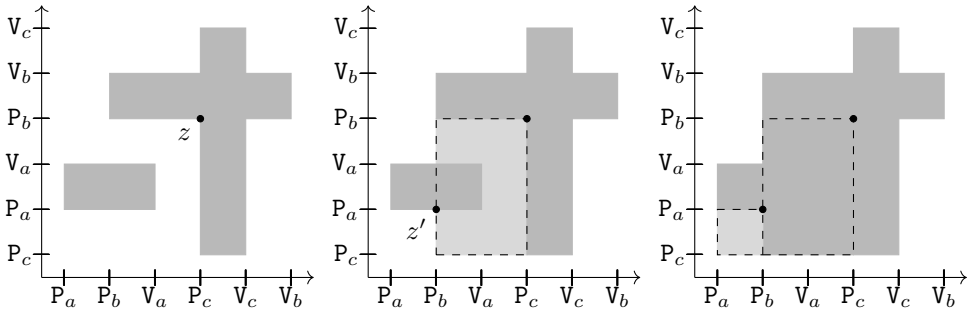
Algorithm 4.1.9. For X a generic d-space of dimension n, as defined in 4.1.1, and a maximal cubical cover $R = \{R^1, \ldots, R^m\}$ of its forbidden region of cardinal m where $R^j = \prod_{i \leq n} [x_i^j, y_i^j]$, the doomed region can be found as follows.

1. Compute all deadlocks $z^1, \ldots, z^p$ using Algorithm 4.1.6.
2. For each deadlock z_k, compute u^k where $u_i^k = \max\{x_i^j \mid j \leq m, x_i^j \neq z_i^k\}$ for all $i \leq n$.
3. Add all cubes $[u_k, z_k]$ to the doomed region.
4. Add all cubes $[u_k, z_k]$ to the forbidden region and reiterate the algorithm until no new cubes are found.

Example 4.1.10. Let us consider the following program

$$P = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_a; \mathtt{P}_c; \mathtt{V}_c; \mathtt{V}_b) \| (\mathtt{P}_c; \mathtt{P}_a; \mathtt{V}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_c)$$

At first there is a single deadlock z, the first round of the algorithm gives us the first part of the doomed region, represented in the middle figure. Adding this new cube to the forbidden region, we get a new "deadlock" z', which generates a new cube of the doomed region. As no more deadlocks exist, the algorithm terminates.



Remark 4.1.11. Unfortunately, this description of the doomed region is flawed: it might intersect the forbidden region as shown in Example 4.1.10, which requires a lot more work to get the correct cubical covering for later uses.

The deadlock detection also works in the case of loops contrary to the previous one, but this simple algorithm fails for the unsafe and doomed regions in the case of programs with even non-nested loops. In [17], a method is proposed to extend this algorithm to programs with loops.

4.1.1.2 Unfolding loops in programs

In [17], Fajstrup defines the *unfolding* of programs with loops. In this section we will present a sightly adapted version of their work, fitted to our previous notation. We only define unfolding for processes of the form $P; L^\star; Q$ where P, Q, L are sequences of instructions (without branchings or loops), but this method extends to multiple loops and even nested loops.

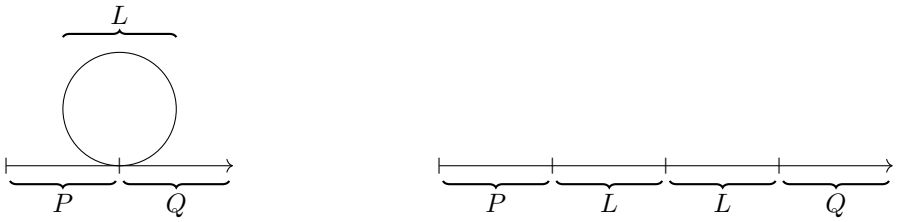
Definition 4.1.12. Let P be a process in a concurrent simple program of the form $L^\star$. We define the *k-unfolding* of P, $\mathcal{U}_k(P)$ as the program where $L^\star$ has be replaced by the sequential composition of n copies of L sequentially composed;

$$\mathcal{U}_k(P; L^\star; Q) = P; L; \ldots; L; Q$$

Remark 4.1.13. The unfolding $\mathcal{U}_-(-)$ easily extends to the case where there are multiple loops, with the possibility of giving a different index (number of delooping) for each loop. It can also be extended to deal with nested loops by applying it recursively, with sets of indexes to capture the nested loops. This fundamentally does not change the core interest of the method, whose presentation we want to keep simple.

We recall that the geometric semantics of $L^\star$ is obtained by identifying the initial and terminal point of the geometric realization of L (with the associated quotient topology), and that the sequential composition of two process corresponds to identifying the endpoint of the geometric realization of the first with the starting point of the second.

Example 4.1.14. For a process of the form $P; L^\star; Q$, the geometric semantics will be equivalent to the space on the left. Then, its 2-unfolding will be equivalent to the space on the right.



Furthermore, all copies of L and their geometric realization can be naturally projected onto the geometric realization of $L^\star$ as previously mentioned by identifying the endpoints. This means we can define a projection from an unfolding to the original program.

Definition 4.1.15. Let X the geometric realization of a program $P; L^\star; Q$ and X_k the geometric realization of its k-unlooping $\mathcal{U}_k(P; L^\star; Q)$. The *projection* $\pi_k \colon X_k \to X$ is given by identifying all k copies of L and identifying their endpoints.

This system of projections verifies some nice properties: it sends the forbidden region of the unfolding onto the forbidden region of the base program [17, Lemma 4.7] and lifts dipaths of the base program to a path in one of the k-unlooping [17, Lemma 4.5] but most notably, as consequence of these properties, it sends the deadlocks of the unfolding onto deadlocks of the base. Again, the method finds all points with no future without care for reachability.

Proposition 4.1.16 ([17, Lemma 4.12])**.** *A point $p \in X$ is a deadlock if and only if for all points $x \in \pi_1^{-1}(p)$, x are deadlocks in the 1-unfolding X_1 (Definition 4.1.15).*

Then to find the deadlocks in a program with loops, we can simply unfold once and apply the deadlock detection Algorithm 4.1.6 to the unfolding. Then the projection will give back all deadlocks in the original program.

Algorithm 4.1.17. Given a program $P = P_1 \| \dots \| P_n$ with each P_i a sequential process without branchings and possibly containing loops, the deadlocks of P can be found by the following algorithm:

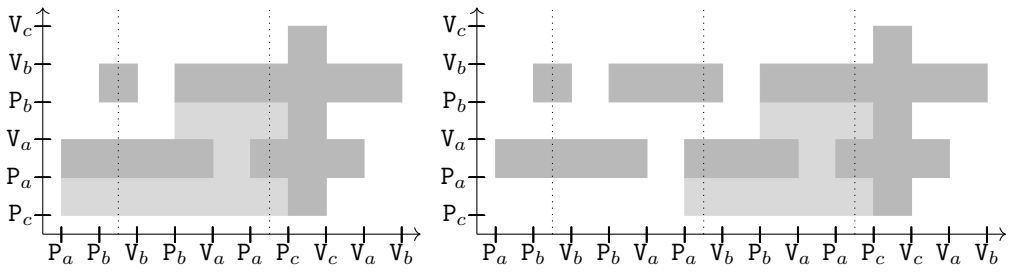
1. Calculate the 1-unfolding $\mathcal{U}_1(P_i)$ of P_i for $i \leq n$
2. Apply Algorithm 4.1.6 to $\mathcal{U}_1(P_1) \| \dots \| \mathcal{U}_1(P_n)$ to get the set of $z_1, \dots, z_m$ deadlocks of $\mathcal{U}_1(P_1) \| \dots \| \mathcal{U}_1(P_n)$.
3. Obtain the deadlocks of P as $\pi_1(z_1), \dots, \pi_1(z_m)$.

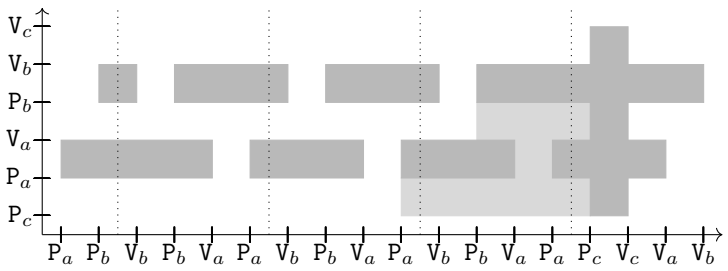
To detect the doomed regions, a similar method can be used by applying Algorithm 4.1.9 to the unfolding, but in this case, a simple 1-unlooping might not suffice as shown in Example 4.1.18. Thankfully, it is shown in [17, Theorem 6.9] that the number of times one need to unfold is finitely bounded in the number of maximal cubes of the original program.

Example 4.1.18. Consider the conservative program $P_1 \| P_2$ with

$$P_1 = (\mathtt{P}_a; \mathtt{P}_b); \mathtt{while}\ b\ \mathtt{do}\ L; (\mathtt{P}_c; \mathtt{V}_c) \qquad L = \mathtt{V}_b; \mathtt{P}_b; \mathtt{V}_a; \mathtt{P}_a$$
$$P_2 = \mathtt{P}_c; \mathtt{P}_a; \mathtt{V}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_c$$

Then if we look at the different successive unfolding of $P_1 \| P_2$, we get the following doomed regions. We see that it actually takes three iterations to find that no parts of the loop L are doomed.





This method has the downside of increasing the size of the forbidden region exponentially in the number of loops in parallel. As Algorithm 4.1.17 scales at least quadratically with its size [19], it is in our best interest to reduce the number of unfolding. For the next part we will switch back to the Algorithm 1.4.33 as it gives much more tractable results, especially for the unsafe and doomed regions. Following the idea explained in this section we will first extend the notion of unfolding to the syntactic models of program. Then we will deepen the projection to work on the cubes of the underlying posets. Lastly we will prove that using our method we require only to unfold the program twice to get the doomed and unsafe regions.

4.1.2 Slight adjustment to the syntactic model

In this section, we go back to the syntactic models of programs of Chapter 3 and define a similar notion of unfolding for those. We use the same grammar for programs as in Definition 3.1.1, but change how we represent the positions in a loop. In Chapter 3, we used the contextual equivalence of Remark 1.1.26 to remove the looping paths and enforce a partial order on the states. This was done at the cost of losing the finiteness of the states in the presence of loops. In the following sections of this chapter, we will go back to considering loops as a pre-order where the terminal and initial positions have been identified.

As we are much more concerned with loops in this chapter, we will first define a few subsets of the programs in **NPIMP**. First let us give a quick reminder of **NPIMP**.

Definition 3.1.1. Let $\mathcal{R}$ be a fixed, finite set of resources. Let $a \in \mathcal{R}$. The language **NPIMP** is generated by the following syntactic expressions, defined by their grammar:

- the set $\mathcal{X}$ of actions:

$$A \quad ::= \quad \mathtt{P}_a \quad | \quad \mathtt{V}_a \quad | \quad \mathtt{skip} \quad | \quad \ldots$$

- the set $\mathcal{C}$ of commands, or programs:

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

We will define three subsets of **NPIMP** that will be of particular interset for this section: the *processes* **Prcs**, consisting of sequential and conditional branchings, the *looping processes* $\mathbf{Prcs}^*$ which are loops of processes, and the *simple loop programs* **Pgrm**, which are the parallel composition of multiple processes. Formally,

Definition 4.1.19. We define, on the language **NPIMP**, with its set of actions $\mathcal{X}$, three sub-languages:

- The set of loop-free processes **Prcs**:

$$S, T \quad ::= \quad A \in \mathcal{X} \quad | \quad S; T \quad | \quad S + T$$

- The set of process and single loops on them $\mathbf{Prcs}^*$:

$$R \quad ::= \quad S \quad | \quad S^\star$$

- The set of *simple loop programs* **Pgrm**:

$$P, Q \quad ::= \quad P \| Q \quad | \quad R$$

4.1.2.1 Looped semantics

In the following, we redefine the semantics of syntactic models to take this modification into account. In order to define the semantics, we will first need to redefine the evaluation/consumption $[\![-]\!]$, the states (positions of Definition 3.1.4) and the reduction relation between states (Definition 3.1.9) starting with the Definition 3.1.4. As the actions of the language have not changed, $[\![-]\!]_{\mathcal{X}}$ remains unchanged. We will then start to redefine the core notions with the new states of the syntactic model, following the presentation of Chapter 3.

Definition 4.1.20. Given a program P, *pre-positions* p are generated by the following grammar:

$$p, q \quad ::= \quad \bot \quad | \quad \top \quad | \quad p ; q \quad | \quad p^\star \quad | \quad p + q \quad | \quad p \between q$$

Where we identify $\bot^\star$ and $\top^\star$ (*i.e.* $\bot^\star = \top^\star$).

Then as in Definition 3.1.7, we extract from the pre-positions, the effective positions (or states) of the program.

Definition 4.1.21. 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:

$$\frac{}{P \vDash \bot} \qquad \frac{P \vDash p}{P ; Q \vDash p ; \bot} \qquad \frac{P \vDash p}{P + Q \vDash p + \varnothing} \qquad \frac{P \vDash p}{P^\star \vDash p^\star}$$

$$\frac{}{P \vDash \top} \qquad \frac{Q \vDash q}{P ; Q \vDash \top ; q} \qquad \frac{Q \vDash q}{P + Q \vDash \varnothing + q} \qquad \frac{P \vDash p \qquad Q \vDash q}{P \between Q \vDash p \between q}$$

We write $\mathcal{P}(P)$ for the set of positions of a program P. When there might be confusion we will write $\bot_P$ (resp. $\top_P$) to indicate $P \vDash \bot$ (resp. $P \vDash \top$).

Now we redefine the reduction relation between the states of Definition 3.1.9 and prove that it preserves the properties we care about (reachability, equivalence with a pre-order)

Definition 4.1.22. The *reduction* relation is defined inductively by the following rules. All the rules not concerning loops remain the same as Definition 3.1.9.

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

$$\frac{}{P{+}Q \vDash \bot \to \bot{+}\varnothing} \qquad \frac{P \vDash p \to p'}{P{+}Q \vDash p{+}\varnothing \to p'{+}\varnothing} \qquad \frac{Q \vDash q \to q'}{P{+}Q \vDash \varnothing{+}q \to \varnothing{+}q'} \qquad \frac{}{P{+}Q \vDash \top{+}\varnothing \to \top}$$

$$\frac{}{P;Q \vDash \bot \to \bot;\bot} \qquad \frac{P \vDash p \to p'}{P;Q \vDash p;\bot \to p';\bot} \qquad \frac{Q \vDash q \to q'}{P;Q \vDash \top;q \to \top;q'} \qquad \frac{}{P;Q \vDash \top;\top \to \top}$$

$$\frac{P \vDash p \to p'}{P \parallel Q \vDash p;q \to p';q} \qquad \frac{Q \vDash q \to q'}{P \parallel Q \vDash p \parallel q \to p \parallel q'}$$

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

While reductions between positions of loops are replaced by the following rules.

$$\frac{}{P^\star \vDash \bot \to \bot^\star} \qquad \frac{P \vDash \bot \to p'}{P^\star \vDash \bot^\star \to p'^\star} \qquad \frac{P \vDash p \to \top}{P^\star \vDash p^\star \to \top^\star} \qquad \frac{}{P^\star \vDash \bot^\star \to \top}$$

$$\frac{P \vDash p \to p' \qquad p, p' \notin \{\bot, \top\}}{P^\star \vDash p^\star \to p'^\star}$$

Lemma 4.1.23. *If $P \vDash p \to p'$ holds then both $P \vDash p$ and $P \vDash p'$ hold.*

Proof. We refer to the proof of Lemma 3.1.11. The new reductions for loops do not change the core argument of the proof. □

Our model is no longer a partial order (because of the identification of $\bot^\star$ and $\top^\star$), but it still keeps the properties of being a well-quasi-order, although it is no longer useful, as we are dealing with finite sets and thus always finitely complemented (Definition 3.2.25).

Definition 4.1.24. Given a program P, the *state space* $\mathcal{G}_P$ of this program is the graph whose vertices are the positions of P (Definition 3.1.7) and edges are the reductions (Definition 3.1.9).

Definition 4.1.25. Given a program P, a *path* on $\mathcal{G}_P$ is a sequence of reduction $\pi = (P \vDash p_i \to p_{i+1})_{0 \le i < n}$ also written $P \vDash \pi : p \to_0^* p_n$ or $\pi \colon p_0 \to^* p_n$. With the following constructions:

- Given two paths $\pi \colon p \to^* q$ and $\pi' \colon q \to^* p'$ we write $\pi' \cdot \pi \colon p \to^* p'$ for their concatenation.
- The empty path on a state p is written $\varepsilon_p : p \to^* p$.

Furthermore, we say that

- An execution trace π of P is a path $P \vDash \pi : \bot \to_P^* p$. When $p = \top_P$, i.e. $P \vDash \pi : \bot \to_P^* \top_P$, we way that π is *total*.
- An execution trace π of P is *maximal* when it cannot be extended.
- An execution is *elementary* when it consists of one reduction step.

- A position p is reachable when there exists an execution trace π with p as target i.e. $P \vDash \pi : \bot \to^*_P p$.

Lemma 4.1.26. *Every position p of P is reachable for $\to^*$*

Proof. See the proof of Lemma 3.1.20. □

As customary, we also write $P \vDash p \to^* p'$ (or sometimes simply $p \to^* p'$) when there exists a path $P \vDash \pi : p \to^* p'$. As before, consumption is equal to evaluation and is defined in the exact same way as for our previous model in Definition 3.1.14.

Definition 4.1.27. Given an execution $P \vDash \pi : p \to^* p'$, we write $[\![P \vDash \pi : p \to^* p']\!] : \Sigma \to \Sigma$ (or sometimes simply $[\![\pi]\!]$) for its *evaluation* or *consumption* of mutexes. This function is defined for elementary executions on each state $\sigma \in \Sigma$ by:

- For $a \in \mathcal{R}$, $[\![\mathtt{P}_a \vDash \pi : \bot \to \top]\!](\sigma) = [\![\mathtt{P}_a]\!]_{\mathcal{X}} = \delta^+_a(\sigma)$,

$$\delta^+_a(\sigma)(b) = \begin{cases} \sigma(a) + 1 & \text{if } b = a \\ \sigma(a) & \text{otherwise} \end{cases}$$

- For $a \in \mathcal{R}$, $[\![\mathtt{V}_a \vDash \pi : \bot \to \top]\!] = [\![\mathtt{V}_a]\!]_{\mathcal{X}} = \delta^+_a(\sigma)$,

$$\delta^+_a(\sigma)(b) = \begin{cases} \sigma(a) - 1 & \text{if } b = a \\ \sigma(a) & \text{otherwise} \end{cases}$$

- for each reduction π, different from the two above, deduced with a rule of Definition 4.1.22 with no premise we have $[\![\pi]\!](\sigma) = \sigma$,
- for each reduction π deduced with a rule of Definition 4.1.22 with one reduction π' as premise, we have $[\![\pi]\!] = [\![\pi']\!]$,

and extended as a morphism of the category of executions on the monoid of consumptions, i.e. $[\![\varepsilon]\!] = \mathrm{id}$ and $[\![\pi \cdot \pi']\!] = [\![\pi']\!] + [\![\pi]\!]$.

The notion of conservative program remains unchanged.

Definition 3.1.15. A program P is *conservative* when for any pair of paths $\pi, \tau \colon x \to y$ in G_P with same source and target we have for any state $\sigma \in \Sigma$ and any resource $a \in \mathcal{R}$:

$$[\![\pi]\!](\sigma)(a) = [\![\tau]\!](\sigma)(a)$$

As before this implies that we can define consumption on positions instead of paths.

Definition 4.1.28. Given a conservative program P and a position p in $\mathcal{P}(P)$, we define the consumption of p, written $[\![p]\!] \colon \mathcal{R} \to \mathbb{Z}$ as follows:

$$[\![p]\!] = [\![\pi]\!](\sigma_0)$$

for some path $\pi \colon \bot_P \to p$ and σ_0 the initial state of P from Definition 3.1.2.

The consumption of a program stays the same as in Chapter 3.

Definition 4.1.29. The *consumption* of a program P is the partial function $\Delta(P) : \mathcal{R} \to \mathbb{Z}$ defined by induction on P by

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

$$\Delta(P;Q) = \Delta(P \parallel Q) = \Delta(P) + \Delta(Q)$$
$$\Delta(\texttt{if } b \texttt{ then } P \texttt{ else } Q) = \Delta(P) \quad \text{if } \Delta(P) = \Delta(Q)$$
$$\Delta(\texttt{while } b \texttt{ do } P) = \underline{0} \quad \text{if } \Delta(P) = \underline{0}$$

where $\underline{0}$ is the constant function equal to 0 and δ_a the indicator function of a.

And the following proposition still holds

Proposition 4.1.30. *The function Δ is only partially defined on programs. A program P is conservative if and only if $\Delta(P)$ is well-defined.*

4.1.2.2 Properties of the state space

As we have loops in our state space, we can no longer hope to have a partial order. But, the state space $\mathcal{P}(P)$ of a program P still retains a pre-order relation that is equivalent to the reduction relation.

Definition 4.1.31. We write $\leq$ for the smallest reflexive relation on the positions of P such that

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

$$\frac{}{p \leq \top} \qquad \frac{p \leq p'}{p+\varnothing \leq p'+\varnothing} \qquad \frac{q \leq q'}{\varnothing+q \leq \varnothing+q'}$$

Proposition 4.1.32. *Given two positions p and p' of P, we have $p \leq p'$ if and only if $p \to^* p'$.*

Proof. This is proved by induction on P.

- P is not of the form $L^\star$. We can refer to the proof of Proposition 3.1.26, as for this case, the reduction relation and induction rules for $\leq$ remain the same.
- $P = L^\star$. Let us suppose $p, p' \notin \{\bot, \top\}$, i.e. $p = l^* \leq l'^* = p'$. Then the inference rules for $\leq$ give:

  $$\frac{}{l^* \leq l'^*}$$

  This implies that $p \leq p'$ always hold. Let us show that it is similar for $p \to^* p'$. By inference rules $l \leq_L \top$ and $\bot \leq_L l'$, i.e. $l \to^* \top$ and $\bot \to^* l'$. Then from the following inference rules

  $$\frac{L \vDash p \to^* \top}{L^\star \vDash p^* \to^* \top^*} \qquad \frac{L \vDash p \to p' \qquad p, p' \notin \{\bot, \top\}}{L^\star \vDash p^* \to p'^*} \qquad \frac{L \vDash \bot \to^* p'}{L^\star \vDash \bot^* \to^* p'^*}$$

we can deduce:

$$\frac{L \vDash p \to^* \top}{L^* \vDash p^* \to^* \top^*} \qquad \frac{L \vDash \bot \to^* p'}{L^* \vDash \bot^* \to^* p'^*}$$

Thus, for $p, p' \notin \{\bot, \top\}$, we have

$$L^* \vDash p \to^* p'$$

Now if $p = \top$, by inference rules, $p \leq p'$ or $p \to^* p'$ implies $p = p' = \top$. Dually, $p' = \bot$ implies $p = p' = \bot$. The remaining cases can be treated similarly to the case $p, p' \notin \{\bot, \top\}$ above, remarking that $p \leq p'$ always holds, adding the following rules in the construction of the path:

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

□

Proposition 4.1.33. *Let P be a program, then its set of positions $(\mathcal{P}(P), \leq)$ is a bounded preorder, with $\bot$ (resp. $\top$) as minimal (resp. maximal) element. Furthermore, if $P \in$ **Prcs** is a program without loops, then its set of positions $\mathcal{P}(P)$ is a well-ordered lattice.*

Proof. By definition of $\to^*$ as a reflexive and transitive closure, and Proposition 4.1.32. When P is loop-free, we can fall back to the proof of Proposition 3.1.32 to prove that it is indeed a lattice. The fact that $\bot$ is the minimal element can be deduced from Lemma 4.1.26 and the fact that there are no reduction with $\bot$ as a target. The property for $\top$ is proved dually. □

4.1.3 2-unfolding of programs: syntactic version

Following the method in [17] presented in Section 4.1.1.2, we define what it means to unfold the syntactic model of a program. In [17], the n-unfolding of a program L^* is given by sequentially composing n copies $L; \ldots; L$ of a program. Thus, a 2-unfolding of L^* correspond to the sequential constructor $L; L$. For more complex programs, the unfolding is defined inductively as follows:

Definition 4.1.34. The 2-*unfolding* $\mathcal{U}$: **Pgrm** $\to$ **Pgrm** of programs is defined inductively as follows:

$$\begin{array}{lll} \mathcal{U}(P \parallel Q) = \mathcal{U}(P) \parallel \mathcal{U}(Q) & \mathcal{U}(P;Q) = \mathcal{U}(P);\mathcal{U}(Q) & \mathcal{U}(\alpha) = \alpha \\ \mathcal{U}(P^*) = \mathcal{U}(P);\mathcal{U}(P) & \mathcal{U}(P+Q) = \mathcal{U}(P)+\mathcal{U}(Q) & \end{array}$$

Remark 4.1.35. $\mathcal{U}$ is the identity for all programs in **Pgrm** without loops.

By definition, the positions of $L; L$ project naturally onto the positions of L. Indeed, by the inference rules in Definition 4.1.21, a position of $L; L$ is either $\bot$, $\top$, $l; \bot$ or $\top; l$ with $l \in L$. Similarly, the positions of L^*, are either $\bot$, $\top$ or l^*.

From this remark we can define a very natural function, mapping positions of a 2-unfolding $L; L$ onto its base program L^* by sending $l; \bot$ and $\top; l$ to l^*.

Definition 4.1.36. Given a program P, we define its *folding projection* $\Phi_P \colon \mathcal{P}(\mathcal{U}(P)) \to \mathcal{P}(P)$ as follows

$$\begin{array}{lll} \Phi_P(\bot_{\mathcal{U}(P)}) = \bot_P & \Phi_{P^\star}(p;\bot) = \Phi_P(p)^* & \Phi_{P+Q}(p+\varnothing) = \Phi_P(p)+\varnothing \\ \Phi_P(\top_{\mathcal{U}(P)}) = \top_P & \Phi_{P^\star}(\top;p) = \Phi_P(p)^* & \Phi_{P+Q}(\varnothing+q) = \varnothing+\Phi_Q(q) \\ \Phi_{P \parallel Q}(p \parallel q) = \Phi_P(p) \parallel \Phi_Q(q) & & \Phi_{P;Q}(p;q) = \Phi_P(p);\Phi_Q(q) \end{array}$$

Remark 4.1.37. By a simple albeit tedious proof by induction, $\Phi_P \colon \mathcal{P}(\mathcal{U}(P)) \twoheadrightarrow \mathcal{P}(P)$ is surjective.

One very nice property of the folding projection is that, for conservative programs, it will preserve the consumption of resources of a given position (Definition 4.1.28). Meaning that if we can compute the forbidden/authorized region in the 2-unfolding of a program (which is loop-free), its pointwise image by the projection Φ will be the forbidden/authorized region of the base program.

Proposition 4.1.38. *Given a program P and a position in its 2-unfolding $p \in \mathcal{P}(\mathcal{U}(P))$, then*

$$[\![p]\!] = [\![\Phi_P(p)]\!]$$

where $[\![-]\!]$ is the resource consumption of a position defined in Definition 4.1.28.

Proof. Let $P \in \mathbf{Pgrm}$. Let us prove by induction on p that

$$\forall p \in \mathcal{P}(P), [\![p]\!] = [\![\Phi(p)]\!]$$

First let us remark that by Definition 4.1.27 $[\![\bot_P]\!] = 0 = [\![\bot_{\mathcal{U}(P)}]\!] = [\![\Phi_P(\bot_P)]\!]$.

- Let $P = \alpha$, then $\mathcal{U}(P) = P$ and $\Phi = id$.
- Let $P = S+T$.
  - $p = s+\varnothing \in \mathcal{P}(\mathcal{U}(P))$, then

$$\begin{array}{rll} [\![p]\!] = [\![\pi \colon \bot \to^* p]\!] & & \textit{Definition } 4.1.28 \\ \quad = [\![\pi \colon \bot \to^* s+\varnothing]\!] & & \\ \quad = [\![(\bot+\varnothing \to^* s+\varnothing) \cdot (\bot \to^* \bot+\varnothing)]\!] & & \text{Definition } 4.1.27 \\ \quad = [\![\bot+\varnothing \to^* s+\varnothing]\!] + [\![\bot \to^* \bot+\varnothing]\!] & & \textit{Definition } 4.1.27 \\ \quad = [\![\bot \to^* s]\!] + \underline{0} & & \text{Definition } 4.1.27 \\ \quad = [\![s]\!] & & \text{Definition } 4.1.28 \\ \quad = [\![\Phi_S(s)]\!] & & \text{Induction Hypothesis} \\ \quad = [\![\bot \to^* \Phi(s)]\!] & & \text{Definition } 4.1.28 \\ \quad = [\![\Phi_S(\bot) \to^* \Phi_S(s)]\!] & & \\ \quad = [\![\Phi_P(\bot)+\varnothing \to^* \Phi(s)+\varnothing]\!] & & \text{Definition } 4.1.28 \\ \quad = [\![\bot \to^* \Phi(s)+\varnothing]\!] & & \text{Definition } 4.1.28 \\ \quad = [\![\Phi(s)+\varnothing]\!] & & \text{Definition } 4.1.28 \\ [\![p]\!] = [\![\Phi_P(p)]\!] & & \end{array}$$

– Similarly if $p = \varnothing + t$, then $[\![p]\!] = [\![\Phi(p)]\!]$

– $p = \top_{\mathcal{U}(P)}$. Let $\pi \colon \bot_{\mathcal{U}(P)} \to^* \top_{\mathcal{U}(P)}$,

$$\begin{aligned}
[\![\top_{\mathcal{U}(P)}]\!] &= [\![\pi]\!] && \text{Definition 4.1.28}\\
&= [\![\bot_{\mathcal{U}(P)} \to^* \top_{\mathcal{U}(S)} + \varnothing]\!] + [\![\top_S + \varnothing \top_{\mathcal{U}(P)}]\!] \\
&= [\![\top_{\mathcal{U}(S)} + \varnothing]\!] + \underline{0} && \text{Definition 4.1.28}\\
&= [\![\Phi(\top_{\mathcal{U}(S)} + \varnothing)]\!] && \text{By the case above}\\
&= [\![\Phi_S(\top_{\mathcal{U}(S)}) + \varnothing]\!] \\
&= [\![\top_S + \varnothing]\!] \\
&= [\![\bot_{S+T} \to^* \top_S +]\!]\varnothing + [\![\top_S + \varnothing \to^* \top_{S+T}]\!] && \text{Definition 4.1.27}\\
&= [\![\bot_{S+T} \to^* \top_{S+T}]\!] && \text{Definition 4.1.27}\\
&= [\![\top_P]\!] && \text{Definition 4.1.28}\\
[\![\top_{\mathcal{U}(P)}]\!] &= [\![\Phi_P(\top_{\mathcal{U}(P)})]\!]
\end{aligned}$$

- The case $P = S ; T$ is treated in much the same way.

- $P = S \parallel T$.

$$\begin{aligned}
[\![s \parallel t]\!] &= [\![\pi \colon \bot_{\mathcal{U}(S) \parallel \mathcal{U}(T)} \to^* s \parallel t]\!] && \text{Definition 4.1.28}\\
&= [\![\pi \colon \bot \parallel \bot \to^* s \parallel t]\!] && \text{Definition 4.1.27}\\
&= [\![\bot \parallel \bot s \parallel \bot \to^* s \parallel t]\!] && \text{by conservativity}\\
&= [\![\bot \parallel \bot \to^* s \parallel \bot]\!] + [\![s \parallel \bot \to^* s \parallel t]\!] \\
&= [\![\bot \to^* s]\!] + [\![\bot \to^* t]\!] && \text{Definition 4.1.27}\\
&= [\![s]\!] + [\![t]\!] && \text{Definition 4.1.28}\\
&= [\![\Phi(s)]\!] + [\![\Phi(t)]\!] && \text{Induction hypothesis}\\
&= [\![\bot \to^* \Phi(s)]\!] + [\![\bot \to^* \Phi(t)]\!] && \text{Definition 4.1.28}\\
&= [\![\bot \parallel \bot \to^* \Phi(s) \parallel \bot]\!] + [\![\Phi(s) \parallel \bot \to^* \Phi(s) \parallel \Phi(t)]\!] \\
&= [\![\bot \parallel \bot \to^* \Phi(s) \parallel \Phi(t)]\!] && \text{Definition 4.1.27}\\
&= [\![\bot_{S \parallel T} \to^* \Phi(s) \parallel \Phi(t)]\!] && \text{Definition 4.1.27}\\
[\![s \parallel t]\!] &= [\![\Phi(s) \parallel \Phi(t)]\!] && \text{Definition 4.1.28}
\end{aligned}$$

- $P = S^*$. Then $\mathcal{U}(P) = S ; S$. First let's remark that for any conservative program X, by Proposition 4.1.30

$$\Delta(X) = [\![\top_{X^*}]\!] = [\![\top_X]\!] = 0 \tag{4.1}$$

– $p = s ; \bot$. As we consider conservative programs, the consumption is independent of the path chosen (Definition 4.1.28), so we will choose a loop-free path $\pi \colon \bot \to^+ \Phi(p)$ to ensure that we only require the $\dfrac{P \vDash p \to p'}{P^* \vDash p^* \to p'^*}$ rule for

reductions in the loop.

$$\begin{aligned}
[\![p]\!] &= [\![\bot_{\mathcal{U}(P)} \to^+ p]\!] && \text{Definition 4.1.28}\\
&= [\![\bot \to \bot\,;\bot]\!] + [\![\bot_S\,;\bot_S \to^+ s\,;\bot_S]\!] && \text{Definition 4.1.27}\\
&= \underline{0} + [\![\bot_S \to^+ s]\!] && \text{Definition 4.1.27}\\
&= [\![\bot^* \to^+ s^*]\!] && \text{Induction Hypothesis}\\
&= [\![\bot_P \to^+ \bot_S^*]\!] + [\![\bot^* \to^+ s^*]\!] && \text{Definition 4.1.27}\\
&= [\![\bot_P \to^+ \Phi(p)]\!]\\
[\![p]\!] &= [\![\Phi(p)]\!] && \text{Definition 4.1.28}
\end{aligned}$$

– $p = \top\,;s$. As $\Phi(\top\,;s) = \Phi(s\,;\bot)$, we will simply prove

$$[\![\top\,;s]\!] = [\![s\,;\bot]\!]$$

Thus

$$\begin{aligned}
[\![\top\,;s]\!] &= [\![\bot \to^* \bot_S\,;\bot_S]\!] + [\![\bot_S\,;\bot_S \to^* \top_S\,;\bot_S]\!]\\
&\qquad + [\![\top_S\,;\bot_S \to^* \top_S\,;s]\!] && \text{Definition 4.1.27}\\
&= [\![\bot_S \to^* \top_S]\!] + [\![\bot_S \to^* s]\!] && \text{Definition 4.1.27}\\
&= [\![\bot_S \to^* s]\!] && \text{Eq. (4.1)}\\
&= [\![\bot_S\,;\bot_S \to^* s\,;\bot_S]\!] && \textit{Definition } 4.1.27\\
&= [\![\bot_P \to^* s\,;\bot_S]\!] && \text{Definition 4.1.27}\\
[\![\top\,;s]\!] &= [\![s\,;\bot_S]\!] && \text{Definition 4.1.28}
\end{aligned}$$

– $p = \top_{\mathcal{U}(P)}$.

$$\begin{aligned}
[\![\top_{\mathcal{U}(P)}]\!] &= [\![\pi]\!]\\
&= [\![\bot_{\mathcal{U}(P)} \to^* \top_{\mathcal{U}(S)}\,;\top_{\mathcal{U}(S)}]\!]\\
&\qquad + [\![\top_{\mathcal{U}(S)}\,;\top_{\mathcal{U}(S)} \to \top_{\mathcal{U}(P)}]\!]\\
&= [\![\top_{\mathcal{U}(S)}\,;\top_{\mathcal{U}(S)}]\!] + \underline{0} && \text{Definition 4.1.28}\\
&= [\![\Phi(\top_{\mathcal{U}(S)}\,;\top_{\mathcal{U}(S)})]\!] && \text{By the case above}\\
&= [\![\top_S\,;\top_S]\!] + \underline{0}\\
&= [\![\bot_P \to^* \top_S\,;\top_S]\!] + [\![\top_S\,;\top_S \to^* \top_P]\!] && \text{Definition 4.1.27}\\
&= [\![\bot_P \to^* \top_P]\!] && \text{Definition 4.1.27}\\
&= [\![\top_P]\!] && \text{Definition 4.1.28}\\
[\![\top_{\mathcal{U}(P)}]\!] &= [\![\Phi(\top_{\mathcal{U}(P)})]\!]
\end{aligned}$$

□

4.2 Syntactic cubical covers for programs with loops

Although we are now dealing with finite syntactic models, the state space might still be large, and we would still like to have a compact representation of the different regions.

The simplest way to go about it is to reuse the representation of sets of positions by cubical covers from Chapter 3.

The folding projection Φ_P already gives a way of going from the unfolding to the base program and back, and if we only wanted to implement Algorithm 4.1.6 we could stop at that, but we hope that we can extend this projection further to make use of the efficient representation given by cubical covers.

Thus, we want to extend the folding projection Φ_P to a projection $\Psi_P \colon \mathcal{C}(\mathcal{U}(P)) \to \mathcal{C}(P)$, sending the cubes of the unfolding to the base program. This projection Ψ_P should at the very least, send a cube c of $\mathcal{U}(P)$ to a cube $\Psi(c)$ whose underlying support is equal to the projection by Φ of the support of c. That means for any cube c, we should have $[\Psi(c)] = \Phi([c])$.

Indeed, using the fact that the folding projection $\Phi \colon \mathcal{P}(\mathcal{U}(P)) \to \mathcal{P}(P)$ preserves forbidden positions (Proposition 4.1.38), this would imply that covers of the forbidden (resp. forbidden) regions of the unfolding are sent to covers of the forbidden (resp. forbidden) regions of the base program.

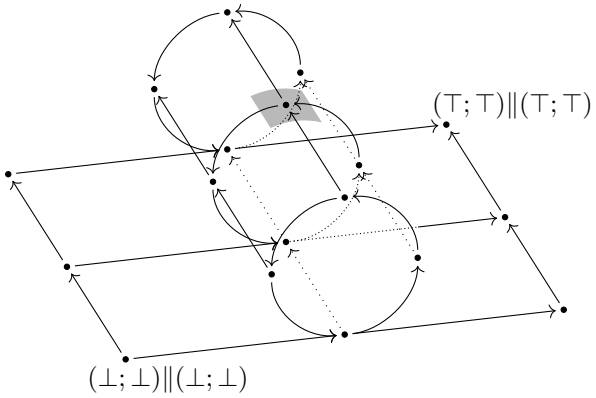
Furthermore, as we are still working with implementation on mind, this projection on cubes should not be too costly to compute.

As cubes are defined by pairs of positions, our first, very natural intuition is that we could use $\Phi \times \Phi \colon \mathcal{P}(\mathcal{U}(P)) \times \mathcal{P}(\mathcal{U}(P)) \to \mathcal{P}(P) \times \mathcal{P}(P)$ as our projection on cubes Ψ_P.

Unfortunately, this approach does not work, but the fault lies not in our choice of projection but more in the definition of our cubes. Indeed, for preorders, as seen in the Example 4.2.1 and Example 4.2.2, there is no way of separating two equivalent elements from being in the same cube. As all elements of a loop are equivalent, this makes it impossible to project any cube onto the loop without increasing the underlying support. In a first time, we should then try to adapt our definition of cubes so that they work in the more general setting of partially ordered sets.

Example 4.2.1. Let us consider the preorder $(\{x, y, z\}, \leq)$ where all elements are equivalent, i.e. $x \leq y \leq z \leq x$. Using Definition 3.2.1, $(x, x) = \{b \mid x \leq b \leq x\} = \{x, y, z\}$. The same is true for all cubes.

Example 4.2.2. Let us consider the program $P = (\mathtt{P}_a; (\mathtt{V}_a; \mathtt{P}_a)^\star) \| (\mathtt{P}_a; \mathtt{V}_a)$ from Example 1.3.33. Its syntactic semantics will give the space below, with the forbidden region in grey.



As the inference rule $\frac{}{p^* \leq p'^*}$ for $\leq$ has no premises, all elements of the loop are equivalent. This means that taking any pair of positions within the loop results in a cube whose support is the whole loop. This implies that the forbidden region, represented in grey, cannot be described by cubes, as any cube trying to describe the loop will have a support equal to the whole section of the loop.

In this section we first introduce the concept of trivial loops with regard to two positions, corresponding to loops that are a "level below" the two positions and can be in a way safely ignored. Then, based on this definition we generalize our previous definition of syntactic cubes to give satisfying results in the case of loops, which are defined as set of points cover by paths without any non-trivial loops. Then, we define our cubical refolding projection Ψ_P by separating which cubes can be projected by $\Phi \times \Phi$ and which cubes need more work, proving that $[\Psi(-)] = \Phi([-])$ in Proposition 4.2.44.

4.2.1 Generalizing syntactic cubes

In this section we will define a new notion of *syntactic cubes* suitable for our new state space for looped programs. Indeed, keeping the same definitions we used for syntactic cubes when the state space was partially ordered leads to some strange results as shown in Example 4.2.2. In most cases, it is impossible to cover a region inside the loop by cubes, which is obviously bad.

Ideally, our new definition of syntactic cube/cubes should be the same as the previous one when considering loop-free programs. Thus, we try to generalize our definition of cubes from Definition 3.2.1.

4.2.1.1 Paths, loops and problems

One can remark that, given a loop-free program P and two positions $p, q \in \mathcal{P}(P)$, the support $[p, q]$ is not only the set of points $\{x \in \mathcal{P}(P) \mid p \leq x \leq q\}$, but also the set of points traversed by a path from p to q (Proposition 3.1.26). This implies that a cube (p, q) can also be defined as the set $\{\pi \colon p \to^* q\}$ while its support would be defined as the union of the "supports" of those paths. This doesn't change any of the properties of our cubes.

Unfortunately, it also does not fix any of the problems for pre-orders. This might be due to the fact, that in our partially ordered setting, any path was loop-free, which for pre-orders is no longer the case. We then give a first attempt at defining our syntactic cubes as set of loop-free paths between two endpoints.

Definition 4.2.3. Let $(\mathcal{P}, \to^*)$ be a finite preorder. Given two elements $p, q \in \mathcal{P}$, a *path* π on $\mathcal{P}$ from p to q is a (possibly empty) finite sequence of reductions $(p_i \to p_{i+1})_{0 \leq i \leq n}$, such that $p = p_0$ and $q = p_{n+1}$.

We write $\pi \colon p \to^* q$ for π is a path from p to q. If π is empty, then $p = q$ and π is called the *empty path* on p, denoted $\varepsilon_P \colon p \to^* p$

Definition 4.2.4. Let $(\mathcal{P}, \to^*)$ be a finite preorder. Given a path $\pi = (p_i \to p_{i+1})_{0 \leq i \leq n}$ on $\mathcal{P}$, we say that π is *loop-free* when for all $0 \leq i \neq j \leq n+1$, $p_i \neq p_j$.

Definition 4.2.5. Let $(\mathcal{P}, \to^*)$ as in Definition 4.2.4. Given two positions $p, q \in \mathcal{P}$, we define the *cube* (p, q) as the sets of all loop-free paths from p to q:

$$(p, q) = \{\pi \mid \pi \colon p \to^* q \text{ with } p, q \in \mathcal{P}, \pi \text{ loop-free}\}$$

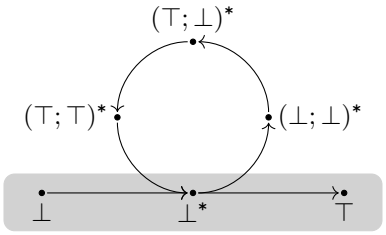
The supports of such cubes is defined as the union of the supports of the paths it contains, i.e. the set of points that are visited by a path.

Definition 4.2.6. Let $(\mathcal{P}, \leq)$ a finite preorder and two elements $p, q \in \mathcal{P}$. Given a path $\pi \colon p \to^* q = (p_i \to p_{i+1})_{0 \leq i \leq n}$,

- for $p' \in \mathcal{P}$, we write $p' \in \pi$, when there exists $0 \leq i \leq n+1$ such that $p' = p'_i$,
- we define the *support* $[\pi]$ of π as $\{p' \in \mathcal{P} \mid p \in \pi\}$

Then the support, i.e. the associated cubical region, of a cube (p, q) from Definition 4.2.5 is defined as $[p, q] = \bigcup_{\pi \in (p,q)} [\pi]$. Once again, this corresponds in the case of loop-free programs, but as shown in Example 4.2.7, the cube $(\bot, \top)$ has some strange behaviour. This suggests that excluding all loops might be a bit too brutal.

Example 4.2.7. Let us consider a very simple program $P = (\texttt{skip}; \texttt{skip})^\star$. Its syntactic semantics is given below, with the support of the cube $(\bot, \top) = \{(\bot \to \bot^* \to \top)\}$ in grey.



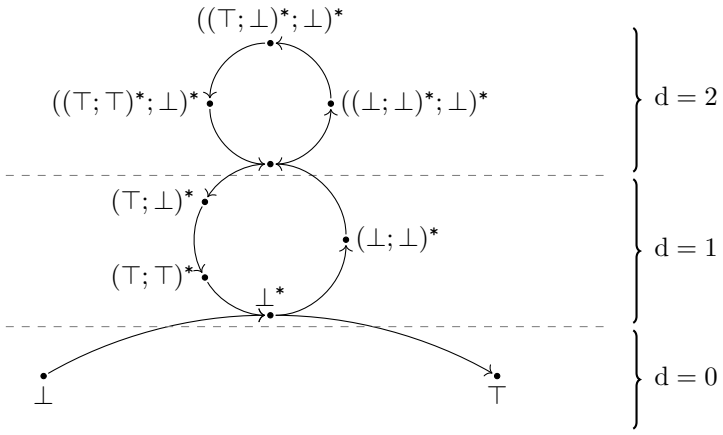
We would expect to be able to describe the whole set of positions simply with the cube $(\bot, \top)$, as in the case of programs without loops. Furthermore, if we were to lift its support with Φ^{-1} we would expect to get the support of the whole set of positions of the unfolding, which would not be the case here.

4.2.1.2 Hierarchy of nested loops

The problem mentioned above comes from the fact that our cubes "see" too much detail in the program. Indeed, in the Example 4.2.7 above, the positions outside (or "above") the loop, care about information of the path in the loop, and are severely restricted as a consequence.

Using this idea that loops of a "smaller" scale than the edges of the cube can be omitted, we will define a notion of *trivial paths* and define cubes as sets of paths with no non-trivial loops, instead of simply excluding all loops.

Example 4.2.8. Let us consider the program $P = ((\texttt{skip};\texttt{skip})^\star;\texttt{skip})^\star$, whose semantics is given below. All positions in the loop are equivalent for the preorder relation. Nonetheless, we would like to distinguish between the level of loop-nesting d of the positions.



We are thus going to equip our preorders with a depth function, to differentiate to which degree positions are equivalent for the preorder relation, this would correspond to checking how many levels of nesting of loops there are for a selected instruction. Sub-paths of a path that only explore position at a greater depth are said to be trivial. The base path should not concern itself with what happens during these sub-paths.

Definition 4.2.9. Given a preorder $\mathcal{P}$ equipped with a function $\mathrm{d}\colon \mathcal{P} \to \mathbb{N}$ called a *depth function*, and a path $\pi\colon p \to^* q$ on elements of $\mathcal{P}$. We say that a sub-path σ of π is d-*trivial* in π if and only if for all $x \in \sigma$, $\mathrm{d}(x) < \mathrm{d}(p)$ and $\mathrm{d}(x) < \mathrm{d}(q)$.

Definition 4.2.10. Given a preorder $\mathcal{P}$, we say that a path π on $\mathcal{P}$ has a *non* d-*trivial loop* when there exists a non-empty sub-path $\sigma \subseteq \pi$ such that σ is not d-trivial and σ is a loop. When a path $\pi\colon p \to^* q$ has no non d-trivial loop, we say that π is *without trivial loops*, and we write $\pi\colon p \to^+ q$.

Definition 4.2.11. Given a sequential program P, we define the *depth* d_P of the position of P as follows.

$$\mathrm{d}_{Q^*}\colon \mathcal{P}(Q^*) \to \mathbb{N}$$
$$p \mapsto \begin{cases} 1 + \mathrm{d}_Q(Q) & \text{if } p = q^* \\ 0 & \text{otherwise} \end{cases}$$

For all other cases

$$\mathrm{d}_{P;Q}(p; \bot) = \mathrm{d}_P(p) \qquad \mathrm{d}_{P+Q}(p + \varnothing) = \mathrm{d}_P(p) \qquad \mathrm{d}_P(\bot) = 0$$
$$\mathrm{d}_{P;Q}(\top; q) = \mathrm{d}_Q(q) \qquad \mathrm{d}_{P+Q}(\varnothing + q) = \mathrm{d}_Q(q) \qquad \mathrm{d}_P(\top) = 0$$

For looped programs, the depth function d is exactly the depth in terms of nested loop needed to reach a position. And a loop is trivial only if it involves positions of a greater depth than the extremities of the path.

For now, we only consider programs without nested loops, so a path π has a non-trivial loop (Definition 4.2.10) if and only if $\pi\colon \bot \to^* \top$ and π has a loop.

Proposition 4.2.12. *Given a program $P \in$ **Pgrm**, and a path $\pi\colon p \to^* q = (p_i \to p_{i+1})_{1 \le i \le n}$ on $\mathcal{P}(P)$, is without non-trivial loops if and only if π is empty or for all $i \neq j$, $p_i = p_j$ implies that p_i is not equivalent to p or q.*

Proof. By definition of d_P and the inference rule $\dfrac{}{p_i^\star \leq q_j^\star}$ two positions p_i of $\mathcal{P}(P^\star)$ are equivalent implies that they have the same depth. Thus, by definition of trivial loops. π is without trivial loops only if $i \neq j$, $p_i = p_j$ implies that p_i is not equivalent to p or q. □

Coming back to Example 4.2.7, all loops are trivial for a path $\pi\colon \bot \to^* \top$, thus the support of the cube $(\bot, \top)$ will give the whole program as expected.

Before we proceed any further we will remark some few interesting properties for paths on the syntactic semantics of a program. First, paths contained in a single iteration of a loop $L^\star$ can be seen as path on the program L, with the property of being without non-trivial loops transferring from one path to another.

Definition 4.2.13. Let $P \in \mathbf{Prcs}$, $p, q \in \mathcal{P}(P)$, $p \leq q$, $p, q \neq \bot, \top$

- Given a path $\pi = (x_i \to x_{i+1})_{i\leq n}\colon p \to^* q$ on positions of P, we define the path $\pi^* = (x_i^\star \to x_{i+1}^\star)_{i\leq n}\colon p^\star \to^* q^\star$ on positions of $P^\star$.
- Given a path $\pi^* = (x_i^\star \to x_{i+1}^\star)_{i\leq n}\colon p^\star \to^* q^\star$ such that for all $i \notin \{0, n+1\}, x_i^\star \neq \top^\star$ we associate the path $\pi^+ = (x_i \to x_{i+1})_{i\leq n}\colon p \to^* q$

Furthermore

- If $\pi\colon p \to^+ q$ then, the associated path $\pi^*\colon p^\star \to^+ q^\star$ is also loop-free
- If $\pi^*\colon p^\star \to^+ q^\star$ then, the associated path $\pi^+\colon p \to^+ q$ is also loop-free.

Proof. (Proof of Definition 4.2.13) Let $P \in \mathbf{Prcs}$, $p \leq q \in \mathcal{P}(P)$, $p, q \neq \bot, \top$. We must prove that the path we construct are indeed valid paths and prove that loop-free properties are transferred.

- Let $\pi = (x_i \to x_{i+1})_{i\leq n}\colon p \to^+ q$ a loop free path. Then by Definition 4.1.22, $\pi^* = (x_i^\star \to x_{i+1}^\star)_{i\leq n}\colon p^\star \to^* q^\star$. Let us suppose the existence of $i \neq j$ such that $x_i^\star = x_j^\star$. As all x_i, x_j are distinct (π loop-free), this implies by Definition 3.1.7, $\{x_i, x_j\} = \{\bot, \top\} \subseteq [\pi]$ This is not compatible with the fact that $p, q \neq \bot, \top$. Thus, π^* has no trivial loops.
- Let $\pi^* = (x_i^\star \to x_{i+1}^\star)_{i\leq n}$ such that $\pi^*\colon p^\star \to^+ q^\star$. $p \leq q$ and for all $i \notin \{0, n+1\}, x_i^\star \neq \top^\star$ implies

  $$i \neq 0, x_i \neq \bot \text{ and for all } i \neq n+1, x_i \neq \top$$

  This implies that for all i, the reduction $x_i^* \to x_{i+1}^*$ is such that $x_i \neq \top$ and $x_{i+1} \neq \bot$. Thus, by Definition 4.1.22, this implies $z_i \to z_{i+1}$. Thus, $\pi = (x_i \to x_{i+1})_{i\leq n}\colon p \to^+ q$. And furthermore, this path is without non-trivial loops as $\mathcal{P}(P)$ is a partial order (Proposition 4.1.33).

□

Furthermore, for certain paths of a looping program $P^\star$, we have a criterion for easily determining if the *core of the loop* $\bot^\star$ belongs to the support. This criterion will be useful later on to distinguish between the cubes of our preorder.

Lemma 4.2.14. *Let $P \in$ **Prcs**. Let $p, q \in \mathcal{P}(P)$ such that $p \leq q$, and $\pi^* = (z_i \to z_{i+1})_{i \leq n} \colon p^\star \to^+ q^\star$ a loop-free path on $\mathcal{P}(P)$. Then for all $i \notin \{0, n+1\}$, $z_i \neq \top^\star$.*

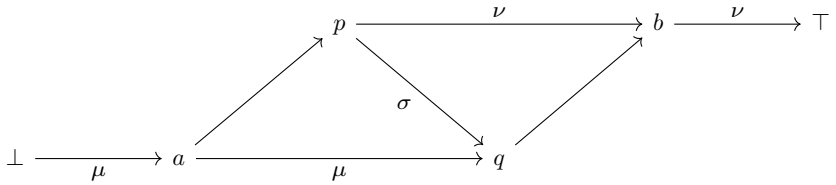
Proof. First, let us remark that if $p^\star$ or $q^\star$ equals $\top^\star$ then the loop-free properties conclude the proof. Now let us suppose a path $\pi^* = (z_i \to z_{i+1})_{i \leq n} \colon p^\star \to^+ q^\star$ that contains no trivial loops such that there exists $0 < j \leq n, z_j = \top^\star \in \pi^*$. Thus, by definition for all $i \neq j, z_i \neq \top^\star$. Furthermore, by definition of the reduction relation for all $i \in I, z_i = x_i^\star$. Then splitting as such

$$\pi^* = \underbrace{(x_i^\star \to x_{i+1}^\star)_{j \leq i \leq n} \colon \bot^\star \to^+ q^\star}_{\mu^*} \circ \underbrace{(x_i^\star \to x_{i+1}^\star)_{i \leq j} \colon p^\star \to^+ \top^\star}_{\nu^*}$$

and applying the Definition 4.2.13 to μ^* and ν^*, we get

- $\nu = (x_i \to x_{i+1})_{i \leq j} \colon p \to^+ \top$
- $\mu = (x_i \to x_{i+1})_{j \leq i \leq n} \colon \bot \to^+ q$
- $\sigma \colon p \to^+ q$ (by Proposition 4.1.32 as $p \leq q$)

Let a (resp. b) the greatest (resp. smallest) element such that there exists a path from a to p and $a \in \mu$ (resp. from q to b and $b \in \mu$). These exist as we have no loops. We get the following



By definition, for $P \in$ **Prcs** the positions a and p (resp. b and q) are the start (resp. end) of a conditional branching. Furthermore, by definition of a, (resp. b), a and q (resp. b and p) are the start and end of the same branching. Hence, we have an inner branching that was opened (at p) inside an outer branching (opened at a) but closed after that the containing branching was closed. This is impossible by construction of the set of positions. □

4.2.1.3 Path-based cubes for preorders

Now, we can finally introduce our new notion of cubes as set of paths without non-trivial loops. Although a great deal of effort was made to reach a satisfying definition, we still need to extend our definition of cubes to consider *composite* cubes. Indeed, even with all our efforts, there are simply too much different cubes in the unfolding to be able to project them all. Composite cubes (and some adjustment to the projection of cubes) are necessary in our model in order to ensure that cubes and their support are projected to the same support, as detailed in Example 4.2.17.

Definition 4.2.15. Let $(\mathcal{P}, \leq)$ a finite preorder. Let $\mathcal{S} \subseteq \mathcal{P}$.

- We call *simple cube* of $\mathcal{P}$ any non-empty subsets of all paths on $\mathcal{P}$ of the following form $(p, q) = \{\pi \mid \pi \colon p \to^+ q \text{ with } p, q \in \mathcal{P}\}$

- For any two cubes (p, q) and (q, r) we call *composite cubes* $(p, q, r) = \{\mu \cdot \nu \mid \nu, \mu \in (p, q) \times (q, r)\}$.

The support $[p, q]$ of the cube (p, q) is defined as

$$[p, q] = \bigcup_{\pi \in (p,q)} [\pi]$$

We write $\mathcal{C}(\mathcal{S})$ the set of all cubes of $\mathcal{P}$ whose support is included in $\mathcal{S}$.

When considering programs, more precisely a looping process $P \in \mathbf{Prcs}^*$ (Definition 4.1.19), we will impose a last restriction on its cubes. For composite cubes, we will only allow composition of cubes of the form $(p, \bot^\star)$ and $(\top^\star, r)$.

Definition 4.2.16. Given, a looped process $P \in \mathbf{Prcs}^*$, the *cubes* of P, written $\mathcal{C}(P)$ are a subset of the cubes of the poset $\mathcal{C}(\mathcal{P}(P))$ containing only cubes of the form:

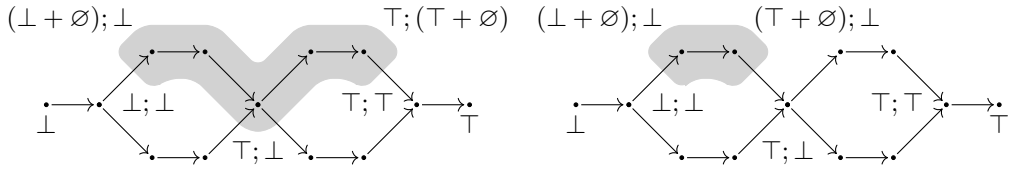
- $(p, q) = \{\pi \mid \pi \colon p \to^+ q\}$ or
- $(p, \bot^\star, q) = (p, \top^\star, q) = \{\pi \mid \pi = \mu \circ \sigma, \sigma \colon p \to^+ \top^\star, \mu \colon \bot^\star \to^+ q\}$

with $p, q \in \mathcal{P}(P)$.

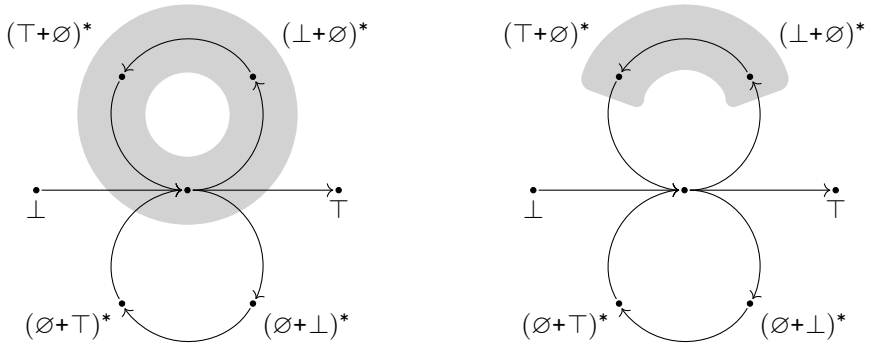
Example 4.2.17. Let us consider the program $P = (\texttt{skip} + \texttt{skip})^\star$ and its unfolding $\mathcal{U}(P) = (\texttt{skip} + \texttt{skip})\texttt{;}(\texttt{skip} + \texttt{skip})$. We need to differentiate between the projections of the two cubes

$$c = ((\bot + \varnothing); \bot, \top; (\top + \varnothing)) \qquad c' = ((\bot + \varnothing); \bot, (\top + \varnothing); \bot)$$

given in grey below, as they should correspond to the lifting of two different cubes, but with the same endpoints.



Then $\Phi([c])$ is covered by $((\bot + \varnothing)^\star, \bot^\star, (\top + \varnothing)^\star)$ and $\Phi([c'])$ is covered by $((\bot + \varnothing)^\star, (\top + \varnothing)^\star)$.



Remark 4.2.18. In Example 4.2.17, one could find a non-composite cube that covers the loop for instance $(\bot^\star, (\top + \varnothing)^\star)$, but then, it would no longer be obtained as a projection by $\Phi \times \Phi$. Furthermore, the maximal cubes of such a region would be too numerous (three in this case).

Remark 4.2.19. Any composition of more than two cubes, is by our restriction equivalent to a composition of only two. Indeed, the composition $(p, q) \cdot (q, r) \cdot (r, s)$ implies $q = r = \top^\star$. Then (q, r) is the set containing only the trivial paths, such that $(p, q) \cdot (q, r) \cdot (r, s) = (p, \top^\star, s)$.

Thankfully most of the composite cubes are actually simple cubes. Most of the remaining composite cubes will those considered in Example 4.2.17.

Lemma 4.2.20. *Let $P \in$ **Prcs**, let $p, q \in \mathcal{P}(P)$. Then*

1. $(\bot, \top^\star, q^\star) = (\bot, q^\star)$
2. $(p^\star, \top^\star, \top) = (p^\star, \top)$
3. $p \not\leq q$ *implies* $(p^\star, q^\star) = (p^\star, \top^\star, q^\star)$

Proof. Let $P \in$ **Prcs**, $p, q \in \mathcal{P}(P)$ such that $p \not\leq q$

1. Let us prove $(\bot, \top^\star, q^\star) = (\bot, q^\star)$
   $P^\star \vDash \bot \to \bot^\star$ is the only reduction possible from $\bot$. Thus, any path $\pi \colon \bot \to^+ p^\star$ is in fact a path $\pi \colon \bot \to^+ \bot^\star \to^+ p^\star$ and conversely.
2. Symmetrically, $(p^\star, \top^\star, \top) = (p^\star, \top)$
3. Now let us prove $(p^\star, q^\star) = (p^\star, \top^\star) \times (\bot^\star, q^\star)$.
   - Let $\pi^* \in (p^\star, q^\star)$. Trivially $\pi^* = (z_i^\star \to z_{i+1}^\star)_{i \in I}$. Then if $\top^\star \notin \pi$, by induction rules, this implies $\pi = (z_i \to z_{i+1} z)_{i \in I}$ is in fact a path from p to q. By Remark 4.2.26 this implies $p \leq q$, which is excluded. Thus, $\top^\star \in \pi$, such that $\pi \colon p^\star \to^+ \top^\star \to q^\star$ i.e.
   $$(p^\star, q^\star) \subseteq (p^\star, \top^\star) \times (\bot^\star, q^\star)$$
   - Now let us suppose $\sigma \cdot \mu \in (p^\star, \top^\star) \times (\bot^\star, q^\star)$. Then if $\sigma \cdot \mu$ contains a non-trivial loop on a point $x^\star$ (which by definition must be distinct from $\top^\star$), this implies $\mu \colon p^\star \to^+ x^\star \to^+ \top^\star$ and $\sigma \colon \bot^\star \to^+ x^\star \to^+ q^\star$. By the same argument as above, this implies $p \leq x \leq q$, which contradicts our hypothesis. Thus
   $$(p^\star, \top^\star) \times (\bot^\star, q^\star) \subseteq (p^\star, q^\star)$$

□

Now we can also define the cube for the parallel composition of processes. This cannot be defined directly in the same manner as for single processes if we hope to achieve consistent result with regard to unfolding as show in Example 4.2.22 below.

Definition 4.2.21. Given a program $P \parallel Q \in \mathbf{Pgrm}$, we define the *cubes* of P as the product of the cubes of P and Q.

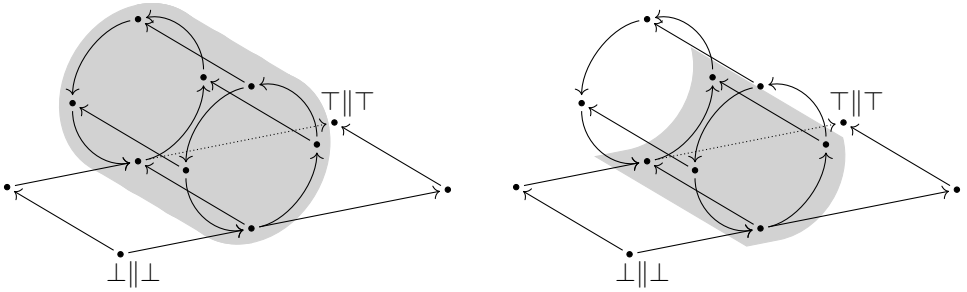
$$\mathcal{C}(P \parallel Q) = \mathcal{C}(P) \times \mathcal{C}(Q)$$

The *support* of such cube $c \times d \in \mathcal{C}(O \parallel Q)$ is then defined as

$$[c \times d] = \{p \parallel q \mid p \in [c], q \in [d]\} = [c] \times [d]$$

Example 4.2.22. Let us consider the program $P = \texttt{skip} \| (\texttt{skip}; \texttt{skip})$, and particularly the cube $c = (\bot \parallel \bot^*, \top \parallel \bot; \bot^*)$

On the left we can see the support of c if we kept the same definition for cubes of parallel processes, and on the right, the support of c with the new definition.



The loops combined with parallel composition, makes it very hard to restrain the paths that can be taken, leading to the huge supports on the right. This would pose a problem, as all cube of the form $(\bot \parallel p^*, \top \parallel q^*)$ would automatically cover all the loop.

Remark 4.2.23. By definition of Φ and Definition 4.2.21, $\Phi([x] \times [y]) = \Phi([x]) \times \Phi([y])$

Remark 4.2.24. If P has no loop, this definition coincides with our previous definition of cubes.

Proposition 4.2.25. *Let $P \in \mathbf{Prcs}^*$. Let $x, y \in \mathcal{P}(P)$, then $x \leq_P y \iff (x, y) \neq \emptyset$ and $(x, \top^*, y) \neq \emptyset$.*

Proof. First let us remark that there exists a finite path $\pi \vDash x \to^* y$ if and only if there exists a path $\pi' \colon x \to^+ y$ without non-trivial loops. Indeed, any finite number of loops can be added or removed to the path without changing the fact that it is a path from x to y. To do this, replace any loop $\sigma \vDash p \to^* p \subseteq \pi$ by the trivial loop ϵ_p.
Then $(x, y) \neq \emptyset$ if an only if there exists $\pi \colon x \to^+ y$. By Proposition 4.1.32 and the remark above this is equivalent to $x \leq y$. □

Remark 4.2.26. For $P \in \mathbf{Prcs}$, $\mathcal{C}(P) = \{(p, q) \mid p, q \in \mathcal{P}(P), p \leq q\}$ and correspond to the cubes defined in Chapter 3. Furthermore, a path $\pi = (z_i \to z_{i+1})_{i \leq n} \colon p \to^+ q$ is a strictly increasing sequence $p = z_0 < z_1 < \cdots < z_{n+1} = q$ of positions of P.

Example 4.2.27. Let $x \in \mathcal{P}(P^*)$ then

- $(x, x) = \{\epsilon_x\}$ and $[x, x] = \{x\}$
- $(\bot, \top)$ is the sets of all paths from $\bot$ to $\top$ and $[\bot, \top] = \mathcal{P}$

4.2.1.4 Order on cubes

As we have changed our definition of cubes, we will also be required to change how we consider the inclusion of such cubes. But in doing so we must ensure that it is coherent with Definition 3.2.4, at the very least, if i is a cube included in j, the support of i should be in j.

Definition 4.2.28. Given a preorder $\mathcal{P}$ and two cubes $i, j \in \mathcal{C}(\mathcal{P})$, we define the relation $\subseteq$ on cubes as follows:

$$i \subseteq j \iff \forall \pi \in i \text{ there exists } \sigma, \mu \text{ such that } \sigma \circ \pi \circ \mu \in j$$

Proposition 4.2.29. *$(\mathcal{C}(\mathcal{P}), \subseteq)$ is a partial order.*

Proof. Let $(p, q) \subseteq (s, t) \subseteq (x, y)$ be three cubes of a program P.

- Reflexivity: Let $\pi \in (p, q)$, then taking ϵ_p (resp. ϵ_q) the trivial loop on p (resp. q), we have, $\epsilon_q \cdot \pi \cdot \epsilon_p \colon p \to^+ q$ i.e. $(p, q) \subseteq (p, q)$.
- Transitivity. Let us suppose $(p, q) \subseteq (s, t) \subseteq (x, y)$.
  Let $\pi \colon p \to^+ q$. Then $(p, q) \subseteq (s, t)$ implies the existence of μ, ν such that $\mu \cdot \pi \cdot \nu \colon s \to^+ t$. Then $(s, t) \subseteq (x, y)$ implies the existence of μ', ν' such that $\mu' \cdot (\mu \cdot \pi \cdot \nu) \cdot \nu' \colon x \to^+ y$. This implies $(\mu' \cdot \mu) \cdot \pi \cdot (\nu \cdot \nu') \colon x \to^+ y$, i.e. $(p, q) \subseteq (x, y)$
- Anti-symmetry. Let us suppose $(p, q) \subseteq (x, y) \subseteq (p, q)$. Let $\pi \colon p \to^+ q$. Then there exists μ, ν, μ', ν' such that $(\mu' \cdot \mu) \cdot \pi \cdot (\nu \cdot \nu') \colon p \to^+ q$. Thus, $\mu' \cdot \mu \colon q \to^+ q$ (resp. $\nu \cdot \nu' \colon p \to^+ p$) implies $\mu' \cdot \mu = \epsilon_q$ (resp. $\nu \cdot \nu' = \epsilon_p$), such that $\mu = \mu' = \epsilon_y = \epsilon_q$ (resp. $\nu = \nu' = \epsilon_x = \epsilon_p$) i.e. $(p, q) = (x, y)$ □

Corollary 4.2.30. *Let $i, j \in \mathcal{C}(\mathcal{P})$, $i \subseteq j$ implies $[i] \subseteq [j]$.*

Definition 4.2.31. Let $\mathcal{P}$ a preorder, we say that a cube i is *maximal* if it is maximal for the inclusion i.e. $\forall j \in \mathcal{C}(\mathcal{P}), i \not\subset j$. Given $X \subseteq \mathcal{P}$, we write $\mathcal{C}^{\max}(R)$ for the set of maximal cubes of X.

4.2.2 Characterizing cubes of the unfolding

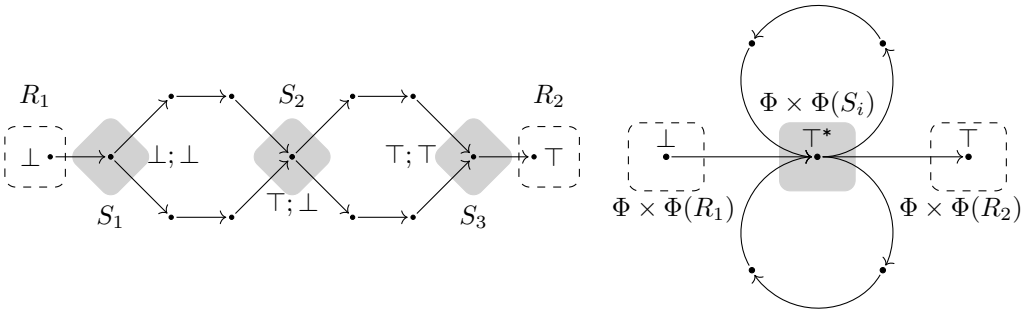
Now that we have a satisfying definition of cubes, we can proceed to define the projection Ψ sending the cubes of the unfolding, onto cubes of the base program. We will first address the case of the unfolding of a single loop $P^\star$.

As we have seen before in Example 4.2.17, we could not simply define cubes as elements of $\mathcal{P}(P^\star) \times \mathcal{P}(P^\star)$, introducing the need for composite cubes. We will need to decide when a cube (x, y) of $\mathcal{U}(P)$ should be sent to a simple cube $(\Phi(x), \Phi(y))$ and when it should be sent to a composite cube $(\Phi(x), \top^\star, \Phi(y))$ instead.

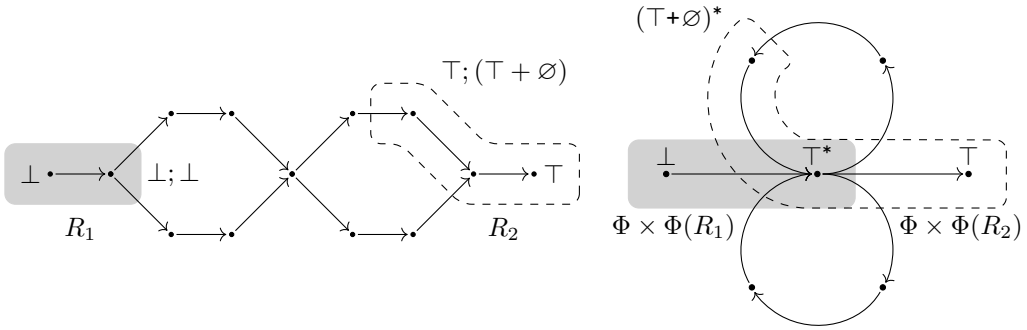
Of course this projection should satisfy some elementary properties, namely, that the support of a cube $\Psi(c)$ should be equal to the image of its support $\Phi[c]$ (Proposition 4.2.44).

As shown below on the unfolding of $P = (\texttt{skip} + \texttt{skip})^\star$, most of the cubes of $\mathcal{U}(P) = (\texttt{skip} + \texttt{skip})\texttt{;}(\texttt{skip} + \texttt{skip})$ do project onto cubes of P via $\Phi \times \Phi$.

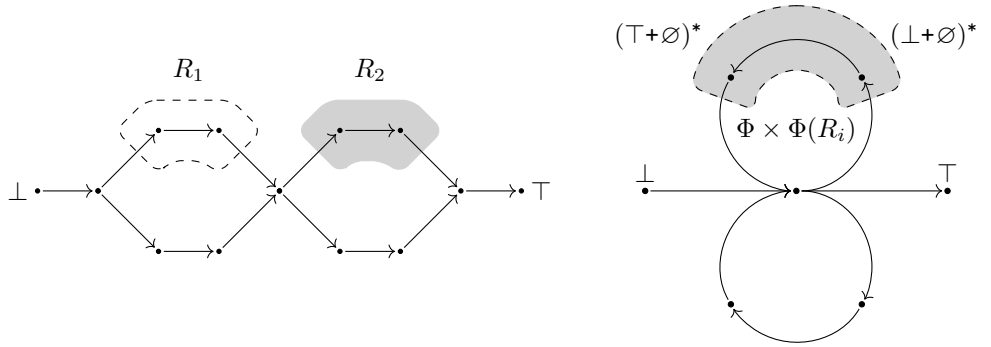
The following singletons all project naturally onto cubes of $P^\star$



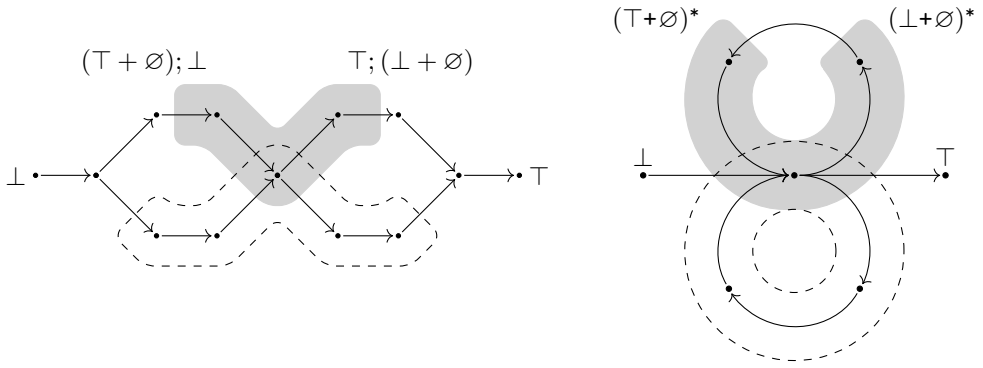
Then, cubes starting at one of the extremities and ending before $\top;\bot$ are sent to cubes starting at one extremity and ending in the loop. It is important that the cubes end before $\top;\bot$ as otherwise, the support would no longer coincide



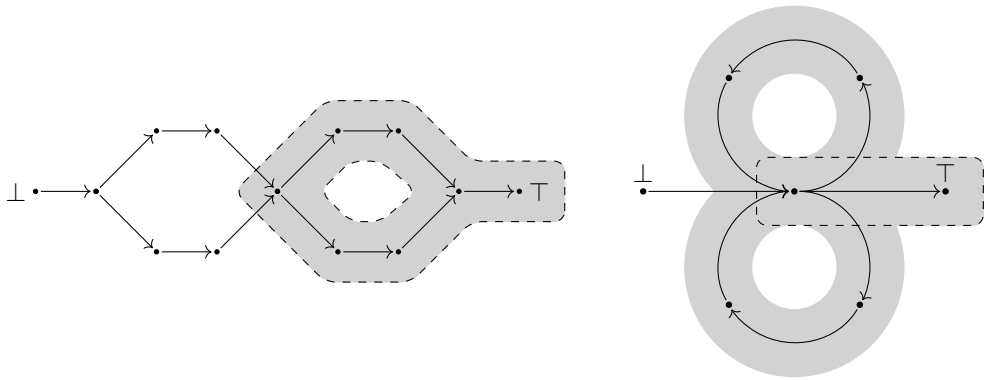
The last type of cubes that are sent via $\Phi \times \Phi$ to simple cubes are the cubes in a single iteration of a loop. These cubes are image of cubes whose endpoints are strictly between $\bot;\bot$ and $\top;\bot$ or strictly between $\top;\bot$ and $\top;\top$.



As suggested by Example 4.2.17 the cubes of the form $(p;\bot, \top;q)$, where $p \leq q$, need to be projected onto composite cube, i.e. to a cube of the form $\Phi(p;\bot) \times \top^* \times \Phi(\top;q)$. When $p \not\leq q$, the composite cube is actually a simple cube (in grey in the figures below) by Lemma 4.2.20. Nonetheless, we will regroup both of these types of cubes going forward as they share many properties.



Some cubes do not have a satisfying way of projecting them, even when considering composite cubes. In the figure below we show such a cube, drawn with the dashed line. Even if we project it on a composite cube, it doesn't cover the projection of its support.



We could change once more our definition to take these into account, but this is unnecessary in our context. These cubes all verify the property that they cover more than a copy of the content of the loop in the unfolding. As we are studying conservative programs, and more precisely maximal covers of the forbidden/authorized regions, we can safely restrict our projection to ignore those cubes without losing anything.

Indeed, by conservativity, the consumption of the loop is null (Proposition 4.1.30). Thus, if the forbidden region covers more than a copy of a loop in the unfolding, this means that it covers the whole program as Φ and Φ^{-1} preserves the consumption of positions.

Thus, we make the choice to not project these cubes, which we can do without loss of generality for the study of programs, and will prove formally in Proposition 4.3.25 that they disappear as we consider maximal covers later on.

We now characterize all cubes of $P^\star$ and its unfolding $\mathcal{U}(P^\star)$. In a first time it allows us to differentiate between cubes that should be sent by Ψ to simple cubes and those that should be sent to composite cubes. It will also prove useful for the various proofs later on, as each of the different subsets have their own unique quirks.

Proposition 4.2.32. *Let $P \in \mathbf{Prcs}^*$, then $\mathcal{C}(P^\star) = \coprod_{i\in[\![-1,5]\!]} \mathcal{L}_i$ where the $\mathcal{L}_i$ are defined below*

$$\begin{aligned}
\mathcal{L}_{-1} &= \{(\bot, \top^\star, \top)\} & \mathcal{L}_2 &= \{(\bot, p^\star) \mid p \in \mathcal{P}(P) \setminus \{\top\}\}\\
\mathcal{L}_0 &= \{(\bot,\bot),(\top,\top),(\bot^\star,\top^\star)\} & \mathcal{L}_3 &= \{(p^\star, \top) \mid p \in \mathcal{P}(P) \setminus \{\bot\}\}\\
\mathcal{L}_1 &= \{(\bot, \top)\} & \mathcal{L}_4 &= \{(p^\star, \top^\star, q^\star) \mid p, q \in \mathcal{P}(P) \setminus \{\top, \bot\}\}
\end{aligned}$$

$$\mathcal{L}_5 = \{(p^\star, q^\star) \mid p \leq q, (p,q) \notin \{(\bot,\bot),(\bot,\top),(\top,\top)\}\}$$

Proof. Given a program $P \in \mathbf{Prcs}$ the positions of $P^\star$ can be separated into the following disjoint sets:

$$\mathcal{P}(P^\star) = \underbrace{\{\bot\}}_{=S_1} \coprod \underbrace{\{\top^\star\}}_{=S_2} \coprod \underbrace{\{p^\star \mid p \in \mathcal{P}(P) \setminus \{\bot,\top\}\}}_{=S_3} \coprod \underbrace{\{\top\}}_{=S_4}$$

We define

$$S_i^j = \{(x,y) \mid x \in S_i, y \in S_j, x \leq y\} \text{ and } L_i^j = \{(x, \top^\star, y) \mid x \in S_i, y \in S_j, x \leq y\}$$

By Proposition 4.2.25 and Definition 4.2.16, we know that

$$\mathcal{C}(P^\star) = \{(p,q) \in \mathcal{P}(P^\star) \mid p \leq q\} \bigcup \{(p, \top^\star, q) \in \mathcal{P}(P^\star) \mid p \leq \top^\star \leq q\}$$

Then by Definition 3.1.25 we have $\bot < p < \top$ and $\bot < q < \top$ for all $p, q \notin \{\bot, \top\}$ such that

$$\mathcal{C}(P^\star) = S \cup L$$

where

$$\begin{aligned}
S &= \Big(\coprod_{i\in[\![1,4]\!]} S_1^i\Big) \coprod \Big(\coprod_{i\in[\![2,4]\!]} S_2^i\Big) \coprod \Big(\coprod_{i\in[\![2,4]\!]} S_3^i\Big) \coprod S_4^4\\
L &= \Big(\coprod_{i\in[\![2,4]\!]} L_1^i\Big) \coprod \Big(\coprod_{i\in[\![2,4]\!]} L_2^i\Big) \coprod \Big(\coprod_{i\in[\![2,4]\!]} L_3^i\Big)
\end{aligned}$$

This makes a lot of cubes to check, we will first proceed by reducing the number of sets to check by remarking that by Lemma 4.2.20, all $L_i^j = S_i^j$ except for L_3^3 and L_1^4, i.e.

$$\mathcal{C}(P^\star) = S \bigcup L_3^3 \bigcup L_1^4$$

Now we separate $S_3^3 = \{(p^\star, q^\star) \mid p, q \notin \{\bot, \top\}\}$ and $L_3^3 = \{(p^\star, \top^\star, q^\star) \mid p, q \notin \{\bot, \top\}\}$ in the following sets

$$\begin{aligned}
S_3^{3\leq} &= \{(p^\star, q^\star) \mid \bot < p \leq q < \top\} & S_3^{3\not\leq} &= \{(p^\star, q^\star) \mid p, q \notin \{\bot,\top\}, p \not\leq q\}\\
L_3^{3\leq} &= \{(p^\star, \top^\star, q^\star) \mid \bot < p \leq q < \top\} & L_3^{3\not\leq} &= \{(p^\star, \top^\star, q^\star) \mid p, q \notin \{\bot,\top\}, p \not\leq q\}
\end{aligned}$$

Such that

$$S_3^3 = S_3^{3\leq} \coprod S_3^{3\not\leq} \qquad\qquad L_3^3 = L_3^{3\leq} \coprod L_3^{3\not\leq}$$

By Lemma 4.2.20 $L_3^{3\not\leq} = S_3^{3\not\leq}$. And by Lemma 4.2.14, every element of $L_3^{3\leq}$ contains a loop, which implies that $S \cap L_3^{3\leq} = \emptyset$. Furthermore, by comparing the supports of their only cube, $L_1^4 \cap S_1^4 = \emptyset$. Thus, we can reduce $\mathcal{C}(P^\star)$ to the disjoint union of the three following sets:

$$\mathcal{C}(P^\star) = S \coprod L_3^{3\leq} \coprod L_1^4 \tag{4.2}$$

By reorganizing the cubes, we get back all the $\mathcal{L}_i$, $i \in [-1:8] \setminus \{6\}$.

- $L_1^4 = \mathcal{L}_{-1}$
- $S_1^1 \cup S_4^4 \cup S_2^2 = \{(\bot, \bot), (\top, \top), (\bot^\star, \bot^\star)\} = \mathcal{L}_0$
- $S_1^4 = \{(\bot, \top)\} = \mathcal{L}_1$
- $S_1^2 \cup S_1^3 = \mathcal{L}_2$. Indeed,

$$\begin{aligned} S_1^2 \cup S_1^3 &= \{(\bot, \bot^\star)\} \cup \{(\bot, p^\star) \mid p \in \mathcal{P}(P) \setminus \{\bot, \top\}\} \\ &= \{(\bot, p^\star) \mid p \in \mathcal{P}(P) \setminus \{\top\}\} \\ S_1^2 \cup S_1^3 &= \mathcal{L}_2 \end{aligned}$$

- Similarly $S_2^4 \amalg S_3^4 = \{(p^\star, \top) \mid p \in \mathcal{P}(P) \setminus \{\bot\}\} = \mathcal{L}_3$
- $L_3^{3\leq} \cup S_3^{3\not\leq} = \mathcal{L}_4$
- As $\bot \leq p \leq \top$ for all $p \in \mathcal{P}(P)$ we have

$$\begin{aligned} S_2^3 \cup S_3^2 \cup S_3^{3\leq} &= \{(p^\star, \top^\star) \mid p \in \mathcal{P}(P) \setminus \{\bot, \top\}\} \\ &\quad \cup \{(p^\star, q^\star) \mid \bot < p \leq q < \top\} \cup \{(\bot^\star, q^\star) \mid q \in \mathcal{P}(P) \setminus \{\bot, \top\}\} \\ &= \{(p^\star, q^\star) \mid (p, q) \notin \{\bot, \top\}^2, p \leq q\} \\ S_2^3 \cup S_3^2 \cup S_3^{3\leq} &= \mathcal{L}_5 \end{aligned}$$

By combining everything in Eq. (4.2), we get $\mathcal{C}(P^\star) = \coprod_{i \in [\![-1,6]\!]} \mathcal{L}_i$ □

Proposition 4.2.33. *Given a program P, then the non-empty cubes of $P;P$, can be split in the following way: $\mathcal{C}(P;P) = \coprod_{i \in [\![-1,6]\!]} \mathcal{C}_i$ where the $\mathcal{C}_i$ are defined below*

$$\begin{aligned} \mathcal{C}_{-1} &= \{(\bot, \top; p) \mid p \in \mathcal{P}(P)\} \cup \{(p; \bot, \top) \mid p \in \mathcal{P}(P)\} \\ &\quad \cup \{(\bot; \bot, \top; \bot), (\top; \bot, \top; \top), (\bot; \bot, \top; \top)\} \\ \mathcal{C}_0 &= \{(\bot, \bot), (\top, \top), (\top; \bot, \top; \bot), (\bot; \bot, \bot; \bot), (\top; \top, \top; \top)\} \end{aligned}$$

$$\begin{aligned} \mathcal{C}_1 &= \{(\bot, \top)\} & \mathcal{C}_4 &= \{(p; \bot, \top; q) \mid p, q \in \mathcal{P}(P) \setminus \{\bot, \top\}\} \\ \mathcal{C}_2 &= \{(\bot, p; \bot) \mid p \in \mathcal{P}(P) \setminus \{\top\}\} & \mathcal{C}_5 &= \{(p; \bot, q; \bot) \mid p \times q \notin \{\bot, \top\}^2, p \leq q\} \\ \mathcal{C}_3 &= \{(\top; p, \top) \mid p \in \mathcal{P}(P) \setminus \{\bot\}\} & \mathcal{C}_6 &= \{(\top; p, \top; q) \mid p \times q \notin \{\bot, \top\}^2, p \leq q\} \end{aligned}$$

Proof. Given a program P, we define the sets

$$S_1 = \{\bot\} \qquad S_2 = \{\bot;\bot\} \qquad S_3 = \{p;\bot \mid p \in \mathcal{P}(P) \setminus \{\bot,\top\}\} \qquad S_4 = \{\top;\bot\}$$
$$S_7 = \{\top\} \qquad S_6 = \{\top;\top\} \qquad S_5 = \{\top;p \mid p \in \mathcal{P}(P) \setminus \{\bot,\top\}\}$$

Such that

$$\mathcal{P}(P;P) = \coprod_{i\in[1:7]} S_i$$

We define $S_i^j = \{(x,y) \mid x \in S_i, y \in S_j, x \leq y\}$. By Proposition 4.2.25,

$$\mathcal{C}(P;P) = \{(p,q) \in \mathcal{P}(P;P) \mid p \leq q\}$$

Then by definition of the order on position (Definition 3.1.25)

$$\mathcal{C}(P;P) = \coprod_{i\leq j} S_i^j$$

- $\mathcal{C}_{-1} = \left(S_1^4 \cup S_1^5 \cup S_1^6\right) \cup \left(S_2^7 \cup S_3^7 \cup S_4^7\right) \cup \left(S_2^4 \cup S_2^6 \cup S_4^6\right)$. Indeed,
  - $S_2^4 \cup S_2^6 \cup S_4^6 = \{(\bot;\bot,\top;\bot),(\bot;\bot,\top;\top),(\top;\bot,\top;\top)\}$
  - $S_1^4 \cup S_1^5 \cup S_1^6 = \{(\bot,\top;q) \mid q \in \mathcal{P}(P)\}$
  - $S_2^7 \cup S_3^7 \cup S_4^7 = \{(p;\bot,\top) \mid p \in \mathcal{P}(P)\}$
- $\mathcal{C}_0 = S_1^1 \cup S_7^7 \cup S_2^2 \cup S_4^4 \cup S_6^6$
- $\mathcal{C}_1 = S_1^7$
- $\mathcal{C}_2 = S_1^2 \cup S_1^3$ Indeed

$$\begin{aligned} S_1^2 \cup S_1^3 &= \{(\bot,\bot;\bot)\} \cup \{(\bot,p;\bot) \mid p \in \mathcal{P}(P) \setminus \{\bot,\top\}\} \\ &= \{(\bot,p;\bot) \mid p \in \mathcal{P}(P) \setminus \{\top\}\} \\ S_1^2 \cup S_1^3 &= \mathcal{C}_2 \end{aligned}$$

- Similarly, $\mathcal{C}_3 = S_5^7 \cup S_6^7$.
- $\mathcal{C}_4 = S_3^5$
- $\mathcal{C}_5 = S_2^3 \coprod S_3^3 \coprod S_3^4$. Indeed,

$$\begin{aligned} S_2^3 \cup S_3^3 \cup S_3^4 &= \{(\bot;\bot,q;\bot) \mid q \notin \{\bot,\top\}\} \cup \{(p;\bot,q;\bot) \mid \bot < p \leq q < \top\} \\ &\qquad \bigcup \{(p;\bot,\top;\bot) \mid p \notin \{\bot,\top\}\} \\ &= \{(p;\bot,q;\bot) \mid (p,q) \notin \{(\bot,\bot),(\bot,\top),(\top,\top)\}, p \leq q\} \\ S_2^3 \cup S_3^3 \cup S_3^4 &= \mathcal{C}_5 \end{aligned}$$

- Similarly $\mathcal{C}_6 = S_4^5 \cup S_5^5 \cup S_5^6$

By combining everything we get $\mathcal{C}(P;P) = \coprod_{i\in[\![-1,7]\!]} \mathcal{C}_i$ □

Now we properly define the sets $\Pi_P\mathcal{L}_i$ (resp. $\Pi_{\mathcal{U}(P)}\mathcal{C}_i$) which are the sets of cubes of a program P (resp. $\mathcal{U}(P)$) that are built inductively using cubes of $\mathcal{L}_i$ (resp. $\mathcal{C}_i$). These will come up often in proofs and lemmas using the inductive structure of our programs.

Definition 4.2.34. Let $P \in \mathbf{Pgrm}$. We define the subsets $\Pi_P\mathcal{L}_i$ of $\mathcal{C}(P)$ and $\Pi_{\mathcal{U}(P)}\mathcal{C}_i$ of $\mathcal{CU}(P)$.

$$\Pi_{\mathcal{U}(P)}\mathcal{C}_i = \begin{cases} \emptyset & \text{if } P \in \mathbf{Prcs} \\ \mathcal{C}_i & \text{if } P = Q^\star \\ (\mathcal{C}(S) \times \Pi_{\mathcal{U}(T)}\mathcal{C}_i) \bigcup (\Pi_{\mathcal{U}(S)}\mathcal{C}_i \times \mathcal{C}(T)) & \text{if } P = S \,\|\, T \end{cases}$$

$$\Pi_P\mathcal{L}_i = \begin{cases} \emptyset & \text{if } P \in \mathbf{Prcs} \\ \mathcal{L}_i & \text{if } P = Q^\star \\ (\mathcal{C}(S) \times \Pi_T\mathcal{L}_i) \bigcup (\Pi_S\mathcal{L}_i \times \mathcal{C}(T)) & \text{if } P = S \,\|\, T \end{cases}$$

As previously stated, the cubes of $\mathcal{C}_{-1} \cap \mathcal{C}(\mathcal{U}(P))$ do not project properly onto cubes of P, and cubes of $\mathcal{L}_{-1}$ cannot be the image of a cube of $\mathcal{U}(P)$. Thus, they will be excluded for now, and later on, we will justify this choice.

Definition 4.2.35. Let $P \in \mathbf{Prcs}$, we define

$$\begin{aligned} \Psi_{P^\star} \colon \mathcal{C}(P;P) \setminus \mathcal{C}{-1} &\to \mathcal{C}(P^\star) \setminus \mathcal{L}{-1} \\ (p,q) &\mapsto \begin{cases} (\Phi(p), \top^\star, \Phi(q)) & \text{if } \ (p,q) \in \mathcal{C}_4 \\ (\Phi(p), \Phi(q)) & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} \Psi_P \colon \mathcal{C}(P) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} &\to \mathcal{C}(P) \setminus \Pi_P\mathcal{L}_{-1} \\ p &\mapsto p \end{aligned}$$

Suppose $P = S \,\|\, T \in \mathbf{Pgrm}$

$$\begin{aligned} \Psi_{S\|T} \colon \mathcal{C}(\mathcal{U}(S \,\|\, T)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} &\to \mathcal{C}(S \,\|\, T) \setminus \Pi_P\mathcal{L}_{-1} \\ s \times t &\mapsto \Psi_S(s) \times \Psi_T(t) \end{aligned}$$

By removing the sets $\mathcal{C}_{-1}$ (already proven to not have an effect) and $\mathcal{L}_{-1}$, Ψ becomes surjective, and we can further restrain the surjection to be compatible with our partition of $\mathcal{C}(P^\star)$ and $\mathcal{C}(\mathcal{U}(P^\star))$.

Proposition 4.2.36. *Let $P \in \mathbf{Prcs}^*$, $\Psi_{P^\star}$ is surjective. More precisely,*

- $\Psi_{P^\star} \colon \mathcal{C}_i \to \mathcal{L}_i$ *is a bijection for* $i \in [\![1,5]\!] \setminus 6$.
- $\Psi_{P^\star} \colon \mathcal{C}_6 \to \mathcal{L}_5$ *is bijective.*
- $\Psi_{P^\star} \colon \mathcal{C}_0 \to \mathcal{L}_0$ *is surjective.*

Proof. Let $P \in \mathbf{Prcs}^*$. Let $p, q \in \mathcal{P}(P)$

- $\Psi_{P^*}\colon \mathcal{C}_0 \to \mathcal{L}_0$ is trivially surjective.
- $\Psi_{P^*}\colon \mathcal{C}_1 \to \mathcal{L}_1$ is trivially bijective.
- $\Psi_{P^*}\colon \mathcal{C}_4 \to \mathcal{L}_4$. First let us remark that

$$p^* = q^* \text{ implies } p = q \text{ or } p, q \in \{\bot, \top\} \tag{4.3}$$

  - Surjectivity: Let $c \in \mathcal{L}_4$, such that $c = (p^*, \top^*, q^*)$, $p, q \notin \{\bot, \top\}$. Then $(p;\bot, \top;q) \in \mathcal{C}_4$ and

$$\Psi(p;\bot, \top;q) = (\Phi(p;\bot), \top^*, \Phi(\top;q)) = (p^*, \top^*, q^*) = c$$

    With $(p;\bot, \top;q) \neq \emptyset$ by the inference rules. Thus, $\Psi_{P^*}\colon \mathcal{C}_4 \to \mathcal{L}_4$ surjective
  - Injectivity: Let $c, d \in \mathcal{C}_4$ such that $c = (p;\bot, \top;q)$, $d = (x;\bot, \top;y)$ with $p, q, x, y \notin \{\bot, \top\}$. Now let us suppose $\Psi(c) = \Psi(d)$.

$$\begin{aligned} \Psi(c) = \Psi(d) &\implies (p^*, \top^*, q^*) = (x^*, \top^*, y^*) \\ &\implies p^* = x^*, q^* = y^* \\ &\implies p = x, q = y \qquad (4.3) \\ \Psi(c) = \Psi(d) &\implies c = d \end{aligned}$$

    Thus $\Psi_{P^*}\colon \mathcal{C}_4 \to \mathcal{L}_4$ injective.

- $\Psi_{P^*}\colon \mathcal{C}_2 \to \mathcal{L}_2$ and $\Psi_{P^*}\colon \mathcal{C}_3 \to \mathcal{L}_3$ follow the same proof.
- $\Psi_{P^*}\colon \mathcal{C}_5 \to \mathcal{L}_5$.
  - Surjectivity: Let $c \in \mathcal{L}_5$, such that $c = (p^*, q^*)$, such that $p \leq q$ with $p, q \neq \bot, \top$. Then $(p;\bot, q;\bot) \in \mathcal{C}_4$ and

$$\Psi(p;\bot, q;\bot) = (\Phi(p;\bot), \Phi(q;\bot)) = (p^*, q^*) = c$$

    With $(p;\bot, q;\bot) \neq \emptyset$ by the inference rules. Thus, $\Psi_{P^*}\colon \mathcal{C}_5 \to \mathcal{L}_5$ surjective
  - Injectivity: Let $c, d \in \mathcal{C}_5$ such that $c = (p;\bot, \top;q)$, $d = (x;\bot, q;\bot)$ with $(p, q), (x, y) \notin \{(\bot, \top), (\bot, \bot), (\top, \top)\}$. Now let us suppose $\Psi(c) = \Psi(d)$.

$$\begin{aligned} \Psi(c) = \Psi(d) &\implies (p^*, q^*) = (x^*, y^*) \\ &\implies p^* = x^*, q^* = y^* \\ &\implies p = x, q = y \qquad (4.3) \\ \Psi(c) = \Psi(d) &\implies c = d \end{aligned}$$

    Thus $\Psi_{P^*}\colon \mathcal{C}_5 \to \mathcal{L}_5$ injective.

- $\Psi_{P^*}\colon \mathcal{C}_6 \to \mathcal{L}_5$ follows the same proof as previous case, using the fact that $\Phi(p;\bot) = \Phi(\top;p)$ for all $p \in \mathcal{P}(P)$.

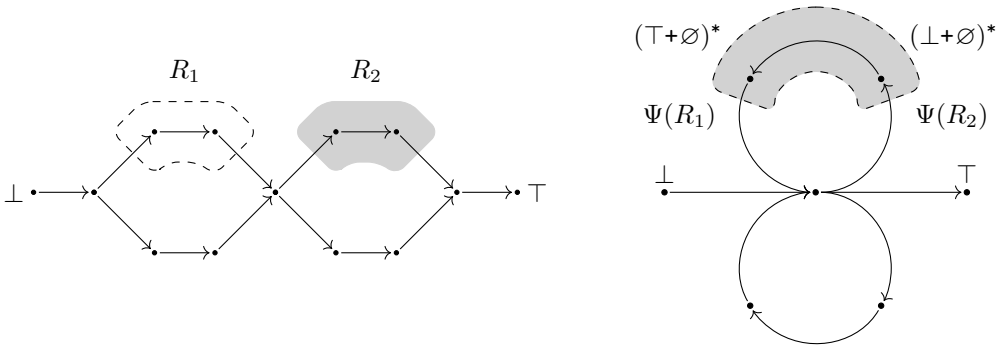
□

Definition 4.2.37. Given $P \in \mathbf{Pgrm}$, we define the equivalence relation $\approx_P$ on cubes of $\mathcal{U}(P)$, such that for all cubes $c, d \in \mathcal{C}(P)$,

$$c \approx_P d \iff \Psi_P(c) = \Psi_P(d)$$

In that case, we say that c, d are *unfold-equivalent*.

Example 4.2.38. Let us consider the program $P = (\texttt{skip+skip})^*$. Below are given the semantics of its unfolding $\mathcal{U}(P)$ and of P respectively on the left and the right. The two cubes R_1 and R_2 are equivalent, as they project on the same cube of P



The example above is actually the only case of equivalent cubes for a simple loop. This implies that two cubes are equivalent if and only if they are both in the lifting of a cube of the base program as we would expect (counting the singleton cube $(\bot^*, \bot^*)$)

Remark 4.2.39. As a follow-up of the previous remark, proved in Proposition 4.2.40 below, we have that Ψ is a discrete fibration, i.e. the antecedents of a cube by Ψ are either equal or their support is disjoint.

Proposition 4.2.40. *Given a program $P \in \mathbf{Pgrm}$, and two cubes $c, c' \in \mathcal{C}(\mathcal{U}(P))$, we have*

$$c \approx c', c \neq c' \implies [c] \cap [c'] = \emptyset$$

Proof. By induction on P.

- $P \in \mathbf{Prcs}$. Nothing to do as Ψ is the identity on programs without loops.
- $P = S \parallel T$. Then $d \in \mathcal{C}(P)$ implies $d = d_1 \times d_2$. Let

  $$c = c_1 \times c_2 \in \Psi^{-1}(d_1) \times \Psi^{-1}(d_2)$$
  $$c' = c'_1 \times c'_2 \in \Psi^{-1}(d_1) \times \Psi^{-1}(d_2)$$

  such that $c \neq c'$. By definition of the support of parallel, this implies that

  $$c_1 \neq c'_1 \text{ or } c_2 \neq c'_2$$

  Let us suppose $c_1 \neq c'_1$. By induction hypothesis, $[c_1] \cap [c'_1] = \emptyset$ i.e. $[c] \cap [c'] = \emptyset$. Similarly, if we suppose $c_2 \neq c'_2$, we have $[c] \cap [c'] = \emptyset$.

- $P = Q^*$. Then for all $c \in \mathcal{C}(P)$, the only cubes that have multiple antecedents by Φ are the cubes of $\mathcal{L}_5$ or $(\bot^*, \bot^*)$.
  - $c = (\bot^*, \bot^*)$ implies $\Psi^{-1}(c) = \{(\bot;\bot, \bot;\bot), (\top;\bot, \top;\bot), (\top;\top, \top;\top)\}$ whose supports are all pairwise disjoint.
  - $c \in \mathcal{L}_5$ and $c = (p^*, q^*) \neq (\bot^*, \top^*)$. Then

    $$\Psi^{-1}(c) = \{(p;\bot, q;\bot), (\top;p, \top;q)\}$$

    As $(p^*, p^*) \neq (\bot^*, \top^*)$, this implies that $p;\bot < \top;\bot < \top;q$ i.e. that the supports of the antecedents of c are disjoints. □

An important property that justifies our choice of projection function Ψ for now is the fact that given a cube c the support of its projection $\Psi(c)$ correspond to the projection by Φ of all positions in the support c. This suggests that we can study the forbidden and authorized region in terms of cubes instead of looking at the set of all positions. We still need to prove that the same holds for Ψ^{-1} and Φ^{-1}, but this is much more complicated.

Theorem 4.2.41. *Let $P \in \mathbf{Pgrm}$, let $c \in \mathcal{C}(\mathcal{U}(P)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$.*

$$\Phi[c] = [\Psi(c)]$$

Proof. By induction on P

- $P \in \mathbf{Prcs}$. All functions are identities.
- $P = P_1 \parallel P_2$. $c \in \mathcal{C}(\mathcal{U}(P)) = \mathcal{C}(\mathcal{U}(P_1) \| \mathcal{U}(P_2))$ implies $c = c_1 \times c_2$. Then by Definition 4.2.21,

$$\begin{aligned}
\Phi([c]) &= \Phi([c_1 \times c_2]) \\
&= \Phi([c_1]) \times \Phi([c_2]) && \text{Remark 4.2.23} \\
&= [\Psi(c_1)] \times [\Psi(c_2)] && \text{Induction hypothesis} \\
\Phi([c]) &= [\Psi(c)]
\end{aligned}$$

- $P = S^*$. Then $\mathcal{U}(P) = S;S$, and using Proposition 4.2.33 we can separate the following cases:
  - $c \in \mathcal{C}_0$. Trivial.
  - $c \in \mathcal{C}_1$. Trivial.
  - $c \in \mathcal{C}_2$. If $x = \bot;\bot$, $\Phi([c]) = \{\bot, \bot;\bot\} = [\Psi(c)]$ and we're done. Now let us suppose $x \neq \bot$

$$\begin{aligned}
\Phi([\bot, x;\bot]) &= \Phi([\bot, \bot;\bot, x;\bot]) \\
&= \Phi(\{\bot, \bot;\bot\}) \bigcup \Phi([\bot;\bot, x;\bot]) \\
&= \{\bot, \bot^*\} \bigcup [\Psi(\bot;\bot, x;\bot)] && \text{By the case } \mathcal{C}_5 \\
&= [\bot, \bot^*] \bigcup [\bot^*, x^*] \\
\Phi([c]) &= [\Psi(c)] && \text{Lemma 4.2.20}
\end{aligned}$$

– $c \in \mathcal{C}_3$. Dual to the case $\mathcal{C}_2$.

– $c \in \mathcal{C}_4$. $c = (x\,;\bot, \top\,;y)$, $\bot < x, y < \top$.

$$\begin{aligned}
\Phi([x\,;\bot,\top\,;y]) &= \Phi([x\,;\bot,\top\,;\bot] \cup [\top\,;\bot,\top\,;y]) \\
&= \Phi([x\,;\bot,\top\,;\bot]) \bigcup \Phi([\top\,;\bot,\top\,;y]) \\
&= [\Psi(x\,;\bot,\top\,;\bot)] \bigcup [\Psi(\top\,;\bot,\top\,;y)] && \text{By the cases } \mathcal{C}_5, \mathcal{C}_6 \\
&= [x^\star, \top^\star] \bigcup [\bot^\star, y^\star] \\
\Phi([c]) &= [\Psi(c)]
\end{aligned}$$

– $c \in \mathcal{C}_5$, $c = (x\,;\bot, y\,;\bot)$ Then

$$\begin{aligned}
v\,;\bot \in [c] &\iff \exists \pi \colon x\,;\bot \to^+ y\,;\bot, \quad v\,;\bot \in \pi \\
&\iff \exists \sigma \colon x \to^+ y, \quad v \in \sigma && \text{Definition 4.1.22} \\
&\iff \exists \sigma^* \colon x^\star \to^+ y^\star, \quad v^\star \in \sigma^* && \text{Definition 4.2.13} \\
&\iff v^\star \in [x^\star, y^\star] \\
v\,;\bot \in [c] &\iff \Phi(v\,;\bot) \in [\Psi(c)]
\end{aligned}$$

– $c \in \mathcal{C}_6$. Similar to the previous case. □

Now we can proceed to define the covers of a given program P. Not much change in this regard: they can still be defined as sets of cubes, and our modifications of the cubes does not affect their definitions.

The fact that we now have a finite preorder does have some implications, notably, there always exists a normal form.

4.2.3 Covers and normal forms

The notion of cover doesn't change from our original definitions. It is still any set of cubes; the order is still as defined in 3.2.11.

Definition 4.2.42. Let $\mathcal{P}$ a preorder. We define a *cover* on $\mathcal{P}$ as a set of cubes of $\mathcal{P}$, and we write $\mathcal{R}(\mathcal{P})$ for the set of all covers of $\mathcal{P}$. The support $[R]$ of a cover R is the union of the support of its cubes i.e. $[R] = \bigcup_{r \in R} [r]$.

We can then naturally extend the definition of the projection Ψ to covers of unfolded program, as well as the equality between support of the projection and projection of the support (Theorem 4.2.41).

Definition 4.2.43. Given a program $P \in \mathbf{Pgrm}$, we define the *projection* on covers of $\mathcal{U}(P)$ as follows:

$$\begin{aligned}
\Psi_P \colon \mathcal{R}(\mathcal{U}(P)) &\to \mathcal{R}(P) \\
\{r_i \mid i \in I\} &\mapsto \{\Psi(r_i) \mid i \in I\}
\end{aligned}$$

This function is only properly defined for covers R such that $R \cap \Pi_{\mathcal{U}(P)} \mathcal{C}_{-1} = \emptyset$.

Proposition 4.2.44. *Given a program $P \in \mathbf{Pgrm}$ and a cover $R \subseteq \mathcal{C}(\mathcal{U}(P)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$, we have*

$$\Phi[R] = [\Psi(R)]$$

Proof. Given a program $P \in \mathbf{Pgrm}$ and a cover $R \subseteq \mathcal{C}(\mathcal{U}(P)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$, we have, for all $c \in R$, $c \notin \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$. Hence, $\Phi([c]) = [\Psi(c)]$ by Theorem 4.2.41. Thus

$$\Phi[R] = \bigcup_{c \in R} \Phi([c]) = \bigcup_{c \in R} [\Psi(c)] = [\Psi(R)]$$

□

We now redefine the order on covers as in Definition 3.2.4. The definition stays the same, as the definition is only dependent on the order defined on the cubes.

As a reminder, two covers are equivalent when they describe the same support. These covers correspond to different ways of describing the same set of points. To correctly manipulate covers it is important to determine when they are equivalent and to maximize efficiency, there should be a most compact form of each region. We recall the 3.2.8 and 3.2.11 of the order on regions.

Definition 4.2.45. Let R and S be two covers of X. The relation $\preceq$ defined as follows is a pre-order on $\mathcal{R}(X)$

$$R \preceq S \iff \forall R^i \in R, \exists S^j \in S, R^i \subseteq S^j$$

with $\subseteq$ defined in Definition 4.2.28

Definition 4.2.46. We can define a partial order $\leq$ on the covers of a partial order X from the preorder $\preceq$ as follows. Given R, S two covers:

$$R \leq S \iff R \preceq S \preceq R \text{ and } S \subseteq R$$

Proposition 4.2.47. *The relation $\leq$ on covers is a partial order.*

Proof. Cf proof of 3.2.12. □

As we are dealing with finite preorders, the normal form Section 3.2.1.3 of a region is always defined. Thus, we no longer need to consider finitely complemented regions (Definition 3.2.27) or normalizable regions as in Chapter 3.

Proposition 4.2.48. *Let R be a region of a finite preoder $\mathcal{P}$. Then R has a maximal cover $N(R)$ called its* normal form *and*

$$N(R) = \mathcal{C}^{\max}(R) = \{c \in \mathcal{C}(R) \mid \forall d \in \mathcal{C}(R), c \not\subset d\}$$

Proof. Given a finite preorder $\mathcal{P}$, the set $\mathcal{P} \times \mathcal{P}$ is also finite. This implies that its set of cubes $\mathcal{C}(\mathcal{P})$ is also finite. Thus, so is the set $\mathcal{C}(R)$. Therefore, the set $\mathcal{C}^{\max}(R)$ is correctly defined as the maximal antichain of the finite partially ordered set $\mathcal{C}(R)$. Then, by definition of $\mathcal{C}^{\max}$ as the maximal antichain, for any cover $C \subseteq \mathcal{R}(R) = \mathfrak{P}(\mathcal{C}(R))$, $C \leq \mathcal{C}^{\max}(R)$. □

4.3 Unfolding conservative covers of loops

As previously stated, we want to show that there is a close link between cubes of the base program and cube of its 2-unfolding in order to perform the deadlock detection algorithm Algorithm 3.3.23 on the unfolding of the program. More precisely, we want to perform the three following steps.

1. We start from cubes of the forbidden region of the base program that we lift to the cubes of forbidden region of its unfolding.
2. Then, we apply Algorithm 3.3.23 on the unfolding, which gives us the deadlocks, as well as unsafe and doomed regions in terms of cubes.
3. Finally, we project these covers back to cubes of the base program.

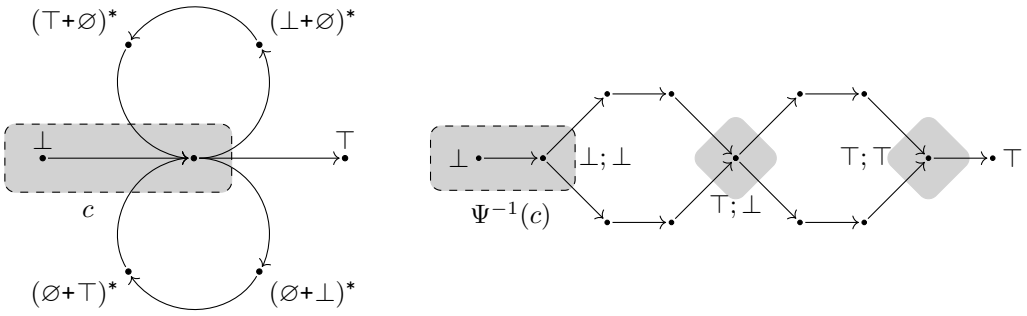
And, at the end of the third step , we hope to recover the cubes of the unsafe/doomed regions.

In order to guarantee this, we need an equivalence between "being in the past" in the unfolding and in the base program for cubes. To prove the correctness of the method above we need the projection and the lifting by Ψ to at least preserve the support of cubes.

The fact that the projection commutes with the support is already proven by Proposition 4.2.44, but proving that the lifting commutes is more complicated.

Indeed, just taking the naive lifting Ψ^{-1} as the inverse image by Ψ does not work on all cubes. Indeed, if we look at the cubes in Example 4.3.1, we can see that lifting cubes does not preserve their supports.

Example 4.3.1. Let us consider $P = (\texttt{skip} + \texttt{skip})^*$, whose semantics is given below. The naive lifting of the cube $(\bot, \top^*)$ does not cover the lifting of its support.



We could instead define the lifting of a cube c as the set $\mathcal{C}^{\max}(\Phi^1[c])$ of all maximal cubes on the lifting of its support. While this works in theory, using it would not be usable in practice, or at least would ruin our efforts of manipulating the regions through their covers.

Instead, we will prove that by considering lifting of covers instead of lifting of cubes we can, under suitable restriction, recover the lifting property.

In this section, we then start by defining a specific subset of regions, the *conservative regions*, which include the forbidden and allowed region of conservative programs. Then,

we prove in Theorem 4.3.15 that when restraining to covers of these conservative regions, the naive lifting Ψ^{-1} preserves the support of the associated region. Finally, we extend this lifting property in Theorem 4.3.27 to prove that the maximal cover of the lifting is actually equal to the lifting of the maximal cover of a conservative region.

4.3.1 Conservative regions and covers

When considering regions of conservative programs, the forbidden (resp. authorized) region can be defined as the set of positions whose consumption is negative (resp. positive). As such, the endpoints of transition that have no effect on the consumption must both be in the same region (forbidden/authorized). We will call these transitions *pure*, and regions that respect these transitions *conservative*. Most of our result depend on the fact that the regions considered are conservative. As the results presented are aimed towards verification of programs, this is not a problem.

Definition 4.3.2. We say a transition is *pure* if it can be obtained inductively using only the following transitions.

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

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

$$\frac{\alpha \neq \mathtt{P}_a, \mathtt{V}_a}{\alpha \vDash \bot \to \top}$$

Definition 4.3.3. Given a program P and $X \subseteq \mathcal{P}(P)$, we say that X is *conservative* when for all $x, y, z \in \mathcal{P}(P)$,

- $x \to y$ is a pure transition implies $x \in X \iff y \in X$

A cover R is said to be conservative when its underlying support $[R]$ is.

Remark 4.3.4. The forbidden and authorized regions of a conservative programs (Definition 1.2.23) are conservative. Indeed, as pure transition do not have any effect on consumption, there is no pure transition between the forbidden region and the authorized region, which are defined by the fact that the consumption differs.

Conservative regions have a lot of useful properties. Notably they form a boolean algebra (Proposition 4.3.6). Furthermore, conservative regions, when split along each process P_i of a program $P_1 \| \dots \| P_n$, preserve their conservativity. This will be very important later on, when manipulating the cubes of parallel processes.

Proposition 4.3.5. *Given a conservative region R on a program P, the region $\Phi^{-1}([R])$ on $\mathcal{U}(P)$ is also conservative.*

Proof. We will prove inductively on the pure transitions of Definition 4.3.2 that for any $x, y \in \mathcal{P}(\mathcal{U}(P))$, $x \to y$ is pure implies $\Phi(x) \to \Phi(y)$ is pure. First, let us check all the base cases:

- For the pure transition $\mathcal{U}(\alpha) \vDash \bot \to \top$, we have $\Phi(\bot) = \bot$, $\Phi(\top) = \top$ such that
$$\alpha \vDash \Phi(\bot) \to \Phi(\top) = \alpha \vDash \bot \to \top$$
which is a pure transition.

- For the pure transition $\mathcal{U}(P{+}Q) \vDash \bot \to \bot{+}\varnothing$, we have

$$P{+}Q \vDash \Phi(\bot) \to \Phi(\bot{+}\varnothing) = P{+}Q \vDash \bot \to \Phi(\bot){+}\varnothing = P{+}Q \vDash \bot \to \bot{+}\varnothing$$

  which is a pure transition.

- The transitions $\mathcal{U}(P{+}Q) \vDash \bot \to \varnothing{+}\bot$, $\mathcal{U}(P{+}Q) \vDash \varnothing{+}\top \to \top$ and $\mathcal{U}(P{+}Q) \vDash \top{+}\varnothing \to \top$ are treated similarly

- The transitions $\mathcal{U}(P;Q) \vDash \bot \to \bot;\bot$, $\mathcal{U}(P;Q) \vDash \top;\top \to \top$ are also treated similarly.

- For the pure transition $\mathcal{U}(P^*) \vDash \bot \to \bot;\bot$, we have $\Phi(\bot) = \bot$, $\Phi(\bot;\bot) = \bot^*$, such that

$$P^* \vDash \Phi(\bot) \to \Phi(\bot;\bot) = P^* \vDash \bot \to \bot^*$$

  which is a pure transition.

- The transition $\mathcal{U}(P^*) \vDash \top;\top \to \top$ is treated similarly.

All the inductive cases follow from the inductive structure of Φ, we only present here one case, all remaining others being treated in a very similar manner. Let us look at the case

$$\mathcal{U}(P \parallel Q) \vDash p \parallel q \to p \parallel q'$$

By inference rules, it is pure, if and only if $\mathcal{U}(Q) \vDash q \to q'$ is. Similarly, the following transition

$$P \parallel Q \vDash \Phi(p \parallel q) \to \Phi(p \parallel q') = P \parallel Q \vDash \Phi \parallel (p)\Phi(q) \to \Phi \parallel (p)\Phi(q')$$

is pure if and only if $\mathcal{U}(Q) \vDash \Phi(q) \to \Phi(q')$ is.

By induction hypothesis $\mathcal{U}(P \parallel Q) \vDash p \parallel q \to p \parallel q'$ equivalent to □

Proposition 4.3.6. *The conservative regions on a preorder $\mathcal{P}$ form a boolean algebra.*

Proof. As a subset of $\mathfrak{P}(P)$, we only need to prove that conservative regions are stable by intersection, union and complement. But this arises naturally from the definition of a conservative region. For the complement, it suffices to remark that $x \to y$ is pure implies $x \in [R] \iff y \in [R]$ which is equivalent to $x \in [R]^{\mathrm{c}} \iff y \in [R]^{\mathrm{c}}$. □

Proposition 4.3.7. *Given a program $P = P_1 \| \dots \| P_n \in$ **Pgrm** and $R = (\prod_{i=1}^n c_i^j)_{j \in J}$ a maximal conservative cover on P, we have that for all $i \in I$ and all $j \in J$:*

$$\pi_i R_{c^j} = \{c \in \mathcal{C}(P_i) \mid c_1^j \times \dots \times c \times \dots \times c_n^j \in R\}$$

are maximal conservative and $c_i^j \in \pi_i R_{c^j}$

Proof. Given a maximal cover $R = (\prod_{i=1}^n c_i^j)_{j \in J}$ on $P_1 \| \dots \| P_n$ and $j \in J$, let us prove that $\pi_i R_{c^j} = \{c \in \mathcal{C}(P_i) \mid c_1^j \times \dots \times c \times c_n^j \in R\}$ is maximal and conservative.

- First let us prove that it is indeed maximal. Let us suppose a cube $d \in \mathcal{C}^{\max}[\pi_i R_{c^j}]$, such that there exists $c \in \pi_i R_{c^j}$, $c \subseteq d$. This implies

$$c_1^j \times \cdots \times c \times \cdots \times c_n^j \subseteq c_1^j \times \cdots \times d \times \cdots \times c_n^j$$

  Furthermore, by definition of $[\pi_i R_{c^j}]$, $c_1^j \times \cdots \times d \cdots \times \times c_n^j \in \mathcal{C}[R]$. Then, by maximality of R,

$$c_1^j \times \cdots \times c \times \cdots \times c_n^j = c_1^j \times \cdots \times d \times \cdots \times c_n^j$$

  i.e. $c = d$. Thus, $\pi_i R_{c^j}$ is indeed maximal.

- Now let us prove that it is conservative. To simplify the proof, we will first prove that the set

$$\pi_i R_p = \{q \in P_i \mid p_1 \| \ldots \| q \| \ldots \| p_n \in [R]\}$$

  is conservative for any $p = p_1 \| \ldots \| p_n \in \mathcal{P}(P)$. By contradiction, let us suppose that there exists $q \in [\pi_i R_p]$ and $q' \notin [\pi_i R_p]$ such that $q \to q'$ is pure. Then, by Definition 4.3.2, the following transition is pure

$$p_1 \| \ldots \| q \| \ldots \| p_n \to p_1 \| \ldots \| q' \| \ldots \| p_n$$

  But, by definition of q', $p_1 \| \ldots \| q' \| \ldots \| p_n \notin [R]$. This contradicts the conservativity of R. Thus, $\pi_i R_p = \{q \in P_i \mid p_1 \| \ldots \| q \| \ldots \| p_n \in [R]\}$ is conservative. Then as

$$\pi_i R_{c^j} = \bigcup_{p \in c^j} \pi_i R_p$$

  and conservative regions form a boolean algebra (Proposition 4.3.6), $\pi_i R_{c^j}$ is conservative.

□

Corollary 4.3.8. *Given a program $P = P_1 \| \ldots \| P_n \in \mathbf{Pgrm}$ and $R = (\prod_{i=1}^n c_i^j)_{j \in J}$ a conservative cover on P, we have that for all $i \in I$, $\pi_i R = (c_i^j)_{j \in J}$ is a conservative cover on P_i.*

Proof. We remark that $\pi_i R = (c_i^j)_{j \in J} = \bigcup_{c^j \in R} \pi_i R_{c_j}$, which are all conservative by Proposition 4.3.7. Then as conservative regions form a boolean algebra (Proposition 4.3.6), $\pi_i R$ is conservative as a cover of the union of conservative regions. □

We regroup here, a few more properties of conservative regions of programs that will be useful in some proofs later on.

Definition 4.3.9. Given a program P and a position $p \in \mathcal{P}(P)$, we define the *predecessors* and *successors* of x respectively as

$$\mathsf{pred}(p) = \{q \in \mathcal{P}(P) \mid q \to p\} \qquad \mathsf{succ}(p) = \{q \in \mathcal{P}(P) \mid p \to q\}$$

Remark 4.3.10. For $P \in \mathbf{Prcs} \backslash \mathtt{P}_a, \mathtt{V}_a$ with a region R on $P^\star$. Then R conservative implies

- $\mathsf{succ}(x) \cap [R] \neq \emptyset$ implies $\mathsf{succ}(x) \subseteq [R]$

- $\mathsf{pred}(x) \cap [R] \neq \emptyset$ implies $\mathsf{pred}(x) \subseteq [R]$

As by induction rules, the only transition in processes that share an endpoint with another transition are pure.

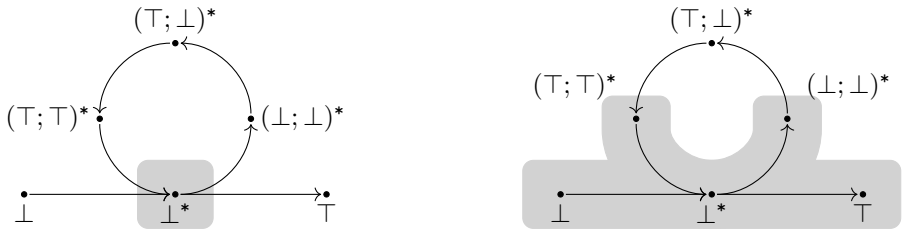
By the remark above, and the definition of pure transition Definition 4.3.2, the core $\top^\star$ is not a conservative region, and will "spread" inside the loop (if the loop is not of the form $\mathtt{P}_a^\star$ or $Va^\star$) and to the extremities of the program $\bot, \top$.

Then, in any maximal cover of such a region, provided the region isn't the whole program, it will always be possible to find a specific subset of cube which will ensure that the cover lifts properly. If the region is the whole program, the maximal cover $\{(\bot, \top)\}$ will also lift properly.

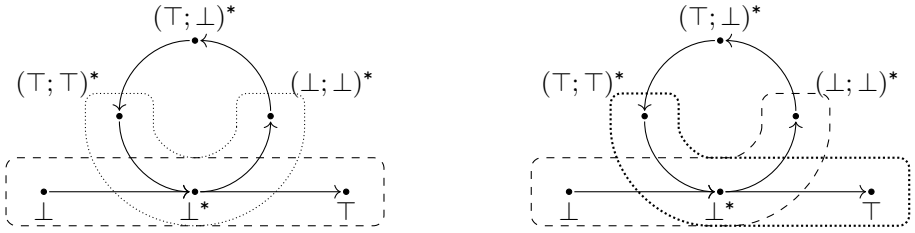
Example 4.3.11. Let us consider the program $P = (\mathtt{P}_a; \mathtt{V}_a)^\star$, whose syntactic semantics is given below. Using the inference rules from Definition 4.1.22, we obtain that the following transition are all pure, as they are obtained using pure transitions.

$$\frac{}{P^\star \vDash \bot \to \bot^\star} \qquad \frac{\overline{P;Q \vDash \bot \to \bot;\bot}}{P;Q \vDash \bot^\star \to \bot;\bot^\star} \qquad \frac{\overline{P;Q \vDash \top;\top \to \top}}{P;Q \vDash \top;\top^\star \to \top;^\star} \qquad \frac{}{P^\star \vDash \bot^\star \to \top}$$

Thus a conservative region on this program containing the core $\top^\star$, as shown on the left actually contains all the positions shown on the right.



And in the maximal cover of such a region, we get the four following cubes:



Lemma 4.3.12. *Given a program $P \in \mathbf{Prcs} \setminus \{\alpha\}$ and R a conservative, maximal cover on $P^\star$ such that $[R] \neq \mathcal{P}(P)$. Then*

$$\bot^\star \in R \implies \exists p, q \in \mathcal{P}(P), p, q \neq \bot, \top \text{ such that } \{(\bot, \top^\star, \top), (\bot, p^\star), (q^\star, \top)\} \subseteq R$$

Furthermore, for any maximal conservative cover R

$$(q^\star, \bot^\star, p^\star) \in R \iff \{(\bot, p^\star), (q^\star, \top)\} \subseteq R$$

Proof. Let $P \in \mathbf{Prcs} \setminus \{\alpha\}$, R a conservative, maximal cover on $P^\star$ such that $[R]^{\mathrm{c}} \neq \emptyset$, i.e. there exists a conservative region $X \subset \mathcal{P}(P)$ such that $R = \mathcal{C}^{\max}(X)$.

- Let us first prove that
$$(\bot, \top^\star, \top) \in R$$
By Definition 4.3.3, $\top^\star \in [R]^{\mathrm{c}}$ implies $\top^\star, \bot, \top \in [R]$, i.e.
$$(\bot, \bot^\star, \top) \in \mathcal{C}[R]$$
As $[R] \neq \mathcal{P}(P)$, $(\bot, \top) \notin \mathcal{C}(R)$. This implies that $(\bot, \bot^\star, \top)$ is maximal in $\mathcal{C}[R] = \mathcal{C}(X)$, i.e.
$$(\bot, \bot^\star, \top) \in \mathcal{C}^{\max}(X) = R$$

- Now let us prove that there exists $p, q \in \mathcal{P}(P), p, q \neq \bot, \top$ such that:
$$(\bot, p^\star), (q^\star, \top) \in R$$
By Remark 4.3.10, $\bot^\star \in [R]$ implies $\mathsf{succ}(\bot^*), \mathsf{pred}(\top^\star) \subseteq [R]$. As $P \neq \alpha$, this implies that the following sets are non-empty
$$X_\bot = \{x \in \mathcal{P}(P) \mid \bot < x < \top, (\bot, x^\star) \in \mathcal{C}(R)\}$$
$$Y_\top = \{x \in \mathcal{P}(P) \mid \bot < x < \top, (y^\star, \top) \in \mathcal{C}(R)\}$$
Let us take p (resp. q) the maximal (resp. minimal) elements of $X_\bot$, (resp. $Y_\top$). Now let us suppose, that there exists a cube $c \in R$ such that, $(\bot, p^\star) \subseteq c$. This implies that
$$c = (\bot, x^\star), \bot < x < \top \text{ or } c = (\bot, \top)$$
As $[R]^{\mathrm{c}} \neq \emptyset$, this implies $(\bot, \top) \notin R$. Thus, $c = (\bot, x^\star), \bot < x < \top$ and $x \in X_\bot$, hence, $x \leq p$ i.e. $c = (\bot, x^\star) \subseteq (\bot, p^\star)$. Such that
$$(\bot, p^\star) \in R$$
By a similar reasoning,
$$(q^\star, \top) \in R$$

- Finally let us prove that
$$(q^\star, \bot^\star, p^\star) \in R \iff \{(\bot, p^\star), (q^\star, \top)\} \subseteq R$$
Let $(\bot, p^\star), (q^\star, \top) \in R$, $p^\star, q^\star \neq \bot$. Then by Lemma 4.2.20, $(\bot^\star, p^\star), (q^\star, \top^\star) \in \mathcal{C}([R])$, which is equivalent, by definition of composite cubes to
$$(q^\star, \top^\star, p^\star) \in \mathcal{C}([R])$$
Now let us prove by contradiction that $(q^\star, \top^\star, p^\star)$ is maximal, and thus in R. Let us suppose $c \in \mathcal{C}^{\max}(R) = R$, such that $(q^\star, \top^\star, p^\star) \subseteq c$. This means that for any pair of paths $(\sigma_1, \sigma_2) \in (q^\star, \top^\star) \times (\top^\star, p^\star)$ and any path $\pi \in c$, there exists paths μ, ν such that
$$\mu \cdot \sigma_1 \cdot \sigma_2 \cdot \nu = \pi$$

As $q^\star, p^\star \neq top^\star$, this implies that π contains $\top^\star$ and that it is not one of its endpoints. By Lemma 4.2.14 that implies that there exists $x, y \in \mathcal{P}(P)$, $\bot < x, y < \top$ and

$$c = (y^\star, \top^\star, x^\star) \in \mathcal{L}_4$$

Furthermore $(q^\star, \top^\star, p^\star) \subseteq (y^\star, \top^\star, x^\star)$ is equivalent to

$$(q^\star, \top^\star) \subseteq (y^\star, \top^\star) \text{ and } (\top^\star, p^\star) \subseteq (\top^\star, x^\star)$$

This is equivalent by Lemma 4.2.20 to

$$(\bot = q^\star)(\bot, \top^\star, q^\star) \subseteq (\bot, \top^\star, y^\star) = (\bot, y^\star)$$
$$(p^\star, \top) = (p^\star, \top^\star, \top) \subseteq (x^\star, \top^\star, \top) = (x^\star, \top)$$

By maximality of $(p^\star, \top)$ and $(\bot = q^\star)$, $p = x$ and $q = y$, which concludes the proof. □

4.3.2 Lifting covers of loops

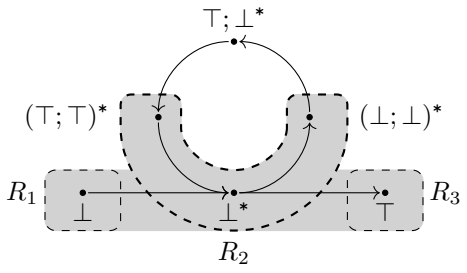
As explained at the beginning of this section, the lifting by Ψ does not preserve the support of cubes, but we prove here that when considering the maximal covers of conservative regions, we can make the naive lifting and the support commute once more.

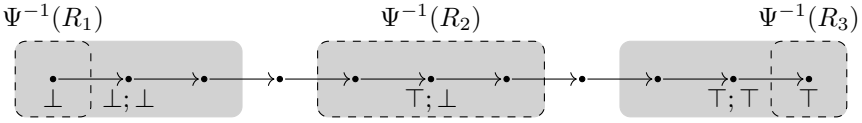
Theorem 4.3.15. *Given a program $P \in$ **Pgrm** and a maximal conservative cover R on P, we have*

$$\Phi^{-1}[R] = [\bigcup_{r \in R} \Psi^{-1}(r)] = [\Psi^{-1}(R)]$$

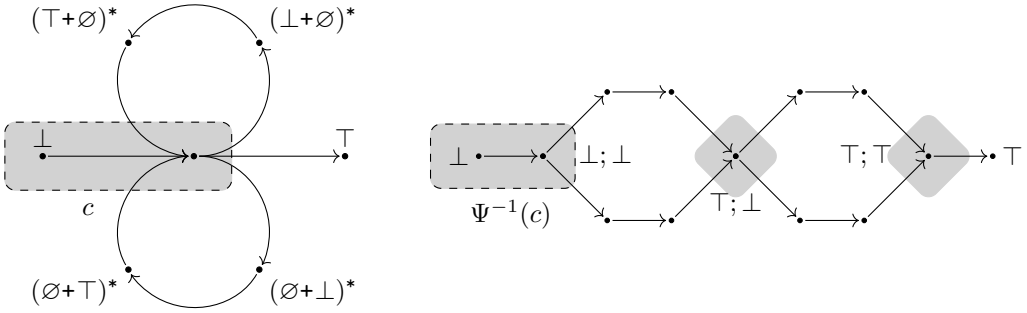
Both the hypothesis of maximality and conservativity are crucial for Theorem 4.3.15 as shown in the examples below.

Let us consider the program $P = (\mathtt{P}_a; \mathtt{V}_a)^\star$ for a resource a of capacity 0, and its unfolding $\mathcal{U}(P) = (\mathtt{P}_a; \mathtt{V}_a); (\mathtt{P}_a; \mathtt{V}_a)$, whose semantics are given below. Let us consider their authorized region in grey (which is conservative by Remark 4.3.23) and a non-maximal cover $R = \{R_1, R_2, r_3\}$. We can see that Theorem 4.3.15 does not hold.



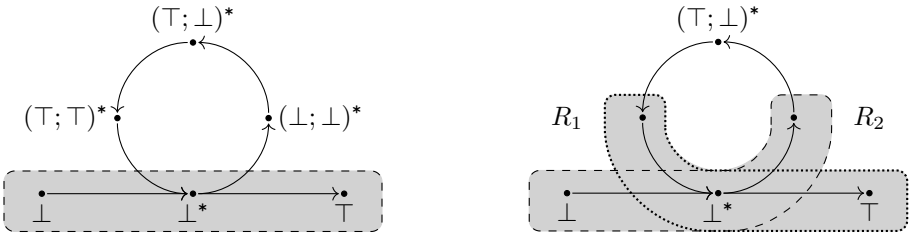


Now let us consider the program $P = (\texttt{skip+skip})^*$ and its unfolding $\mathcal{U}(P) = (\texttt{skip+skip});(\texttt{skip+skip})$ whose semantics are given respectively on the left and right below. Let us consider the non-conservative region X, containing a single maximal cube c, and $\Phi^{-1}(X)$ are given in grey below. Once again, Theorem 4.3.15 does not hold.



As we cannot lift cubes of $\Pi\mathcal{L}_{-1}$ it is important to prove that they do not contribute to the support of a maximal cover. This is a direct consequence of the Lemma 4.3.12. Indeed, the presence of $(\bot, \top^*, \top)$ in a maximal conservative cover implies the existence of other cubes that cover the same support ash shown in Example 4.3.13.

Example 4.3.13. Let us consider the program $P = (\texttt{skip};\texttt{skip})^*$ from Example 4.3.11, whose semantics is given below. The region in grey on the left below is not conservative, and by Lemma 4.3.12, a conservative region X such that $[\bot, \top^*, \top] \subseteq X$ will be of the form on the left, spreading inside the loop.



Thus, a maximal cover of such a region will contain $(\bot, \top^*, \top)$, but also the cubes R_1 and R_2, which cover a bigger region than $(\bot, \top^*, \top)$.

Lemma 4.3.14. *Let R a conservative cover on a program P, such that R maximal. Then*

$$[R] = [R \setminus \Pi_P\mathcal{L}_{-1}]$$

Proof. First let us note that If $R \cap \Pi\mathcal{L}_{-1} = \emptyset$. Then by definition $[R] = [R \setminus \Pi\mathcal{L}_{-1}]$. For the rest of the proof, we can suppose, $R \cap \Pi\mathcal{L}_{-1} \neq \emptyset$. Then by induction on P

- $P \in \mathbf{Prcs}$. Then $\Pi\mathcal{L}_{-1} = \emptyset$. This concludes the proof.
- $P = Q^*$ Otherwise if $(\bot, \bot^*, \top) \in R$. Then $\top^* \in [R]$. By Lemma 4.3.12, this implies that there exists p, q such that $(\bot, q^*), (p^*, \top) \in R$. Furthermore,

$$\{\bot, \bot^*, \top\} \subseteq [\bot, q^*] \cup [p^*, \top] \subseteq [R \setminus \mathcal{L}_{-1}]$$

Thus

$$\begin{aligned}[R] &\subseteq [R \setminus \mathcal{L}_{-1}] \cup [\mathcal{L}_{-1}] \\ &\subseteq [R \setminus \mathcal{L}_{-1}] \cup \{\bot, \bot^*, \top\} \\ &\subseteq [R \setminus \mathcal{L}_{-1}] \cup [R \setminus \mathcal{L}_{-1}] \\ [R] &\subseteq [R \setminus \mathcal{L}_{-1}]\end{aligned}$$

And as the support is increasing, $[R \setminus \mathcal{L}_{-1}] \subseteq [R]$. Thus,

$$[R] = [R \setminus \mathcal{L}_{-1}]$$

- $P = P_1 \parallel P_2$. We define for all $c = c_1 \times c_2 \in R$ the sets $\pi_1 R_c = \{c' \in \mathcal{P}(P_1) \mid c' \times c_2 \in R\}$ and $\pi_1 R_c = \{c' \in \mathcal{P}(P_2) \mid c_1 \times c' \in R\}$. By definition of the sets $\pi_i R_c$, we have

$$R = \bigcup_{c \in R} \pi_1 R_c \times \pi_2 R_c$$

And furthermore, by definition

$$R \setminus \Pi_P \mathcal{L}_{-1} = \bigcup_{c \in R} \pi_1 R_c \setminus \Pi_{P_1} \mathcal{L}_{-1} \times \pi_2 R_c \setminus \Pi_{P_2} \mathcal{L}_{-1}$$

As R is maximal, the Proposition 4.3.7 implies that for all $c \in R$, we have $\pi_1 R_c$ and $\pi_2 R_c$ are maximal and conservative respectively on P_1 and P_2. We can then apply the induction hypothesis on $\pi_i R_c$ for $i = 1, 2$ such that

$$[\pi_i R_c \setminus \Pi_{P_i} \mathcal{L}_{-1}] = [\pi_i R_c]$$

Thus, we get:

$$\begin{aligned}[R] &= [\bigcup_{c \in R} \pi_1 R_c \times \pi_2 R_c] \\ &= \bigcup_{c \in R} ([\pi_1 R_c] \times [\pi_2 R_c]) \\ &= \bigcup_{c \in R} ([(\pi_1 R_c) \setminus \Pi_{P_1} \mathcal{L}_{-1}] \times [\pi_2 R_c \setminus \Pi_{P_2} \mathcal{L}_{-1}]) \\ &= [\bigcup_{c \in R} (\pi_1 R_c) \setminus \Pi_{P_1} \mathcal{L}_{-1} \times \pi_2 R_c \setminus \Pi_{P_2} \mathcal{L}_{-1}] \\ [R] &= [R \setminus \Pi_P \mathcal{L}_{-1}]\end{aligned}$$

□

We can finally prove Theorem 4.3.15, which states that Ψ lifts conservative cover to a cover of their unfolding (in terms of regions). As forbidden regions of programs are conservative (Remark 4.3.4), this implies that we can unfold the forbidden region via its cover instead of lifting each position by Φ^{-1}.

In the following, we will talk interchangeably about $\mathcal{C}^{\max}(\Psi^{-1}(R))$ and $\mathcal{C}^{\max}(\Phi^{-1}[R])$ as they are the same set.

Theorem 4.3.15. *Given a program $P \in \mathbf{Pgrm}$ and a maximal conservative cover R on P, we have*

$$\Phi^{-1}[R] = [\bigcup_{r \in R} \Psi^{-1}(r)] = [\Psi^{-1}(R)]$$

Proof. Let $P \in \mathbf{Pgrm}$ and R a maximal conservative cover. Let us prove the equality by induction on P.

- $P \in \mathbf{Prcs}$. Trivial
- $P = P_1 \parallel P_2$. We define for all $c = c_1 \times c_2 \in R$ the sets $\pi_1 R_c = \{c' \in \mathcal{P}(P_1) \mid c' \times c_2 \in R\}$ and $\pi_1 R_c = \{c' \in \mathcal{P}(P_2) \mid c_1 \times c' \in R\}$. By definition of the sets $\pi_i R_c$, we have

  $$R = \bigcup_{c \in R} \pi_1 R_c \times \pi_2 R_c$$

  As R is maximal, the Proposition 4.3.7 implies that for all $c \in R$, we have $\pi_1 R_c$ and $\pi_2 R_c$ are maximal and conservative respectively on P_1 and P_2. We can then apply the induction hypothesis on $\pi_i R_c$ for $i = 1, 2$. This gives us

  $$\Phi^{-1}(\pi_i R_c) = \Psi^{-1}(\pi_i R_c)$$

  This implies

  $$\begin{aligned}
  \Phi^{-1}[R] &= \Phi^{-1}[\bigcup_{c \in R} \pi_1 R_c \times \pi_2 R_c] \\
  &= \bigcup_{c \in R} \Phi^{-1}[\pi_1 R_c] \times \Phi^{-1}[\pi_2 R_c] \\
  &= \bigcup_{c \in R} [\Psi^{-1}(\pi_1 R_c)] \times [\Psi^{-1}(\pi_2 R_c)] \\
  &= [\Psi^{-1}(\bigcup_{c \in R} \pi_1 R_c \times \pi_2 R_c)] \\
  \Phi^{-1}[R] &= [\Psi^{-1}(R)]
  \end{aligned}$$

- $P = Q^\star$. First let us remark that $[R] = \mathcal{P}(P)$ implies by maximality of R that $R = \{(\bot, \top)\}$, and thus

  $$\Phi^{-1}(R) = \mathcal{P}(\mathcal{U}(P)) = [\bot, \top] = [\Psi^{-1}(R)]$$

  We can thus suppose $[R] \neq \mathcal{P}(P)$ for the rest of this case. We proceed to prove the equality by double inclusion.

  - First let us prove $\Phi^{-1}([R]) \subseteq [\Psi^{-1}(R)]$. We will proceed by separating the cases depending on the value of $x \in [R]$.
    - If $x \in \{\bot, \top, \bot^\star\}$, by conservativity of R, this implies $\bot^\star \in R$. Then, as we suppose $[R] \neq \mathcal{P}(P)$, we can apply Lemma 4.3.12, which implies here exists $p, q \in \mathcal{P}(Q)$, $p, q \neq \bot, \top$ such that

      $$(\bot, p^\star), (q^\star, \top), (q^\star, \top^\star, p^\star) \in R$$

By definition of Ψ, this implies

$$(\bot, p; \bot), (\top; q, \top), (p; \bot, \top; q) \in \Psi^{-1}(R)$$

Thus

$$\Phi^{-1}(x) \subseteq \Phi^{-1}(\{\bot, \top, \bot^*\}) = \{\bot; \bot, \top; \bot, \top; \top\} \subseteq [\Psi^{-1}(R)]$$

∗ Otherwise $x = y^\star$, $y \neq \bot, \top$. Thus,

$$\Phi^{-1}(x) = \{y; \bot, \top; y\}$$

· If $x \in [p^\star, q^\star]$, $(p^\star, q^\star) \in R \cap \mathcal{L}_5$. Then by Definition 4.2.13, $y \in (p, q)$. By inference rules on cubes of loop-free program, this implies

$$y; \bot \in (p; \bot, q; \bot) \qquad \text{and} \qquad \top; y \in (\top; p, \top; q)$$

Thus

$$\Phi^{-1}(x) \subseteq [\{(p; \bot, q; \bot), (\top; p, \top; q)\}] = [\Psi^{-1}(p^\star, q^\star)] \subseteq [\Psi^{-1}(R)]$$

· Otherwise, $x \in [R]$ implies that there is a cube c of R that is not in $\mathcal{L}_5 \cup \mathcal{L}_1$ such that $x \in [c]$. By looking at the supports, $c \in \mathcal{L}_4 \cup \mathcal{L}_4 \cup \mathcal{L}_2 \cup \mathcal{L}_3$, i.e. there exists $p, q \in \mathcal{P}(Q), p, q \neq \bot, \top$ such that

$$c = (\bot, p^\star) \text{ or } c = (q^\star, \top) \text{ or } c = (q^\star, \top^\star, p^\star)$$

By Lemma 4.3.12, if one of these cubes belongs to R, then we can choose p or q such that they all belong to R. By definition of $(q^\star, \top^\star, p^\star)$

$$x \in (q^\star, \top^\star, p^\star) \text{ and } \Big(x \in (\bot, p^\star) \text{ or } (q^\star, \top)\Big)$$

Let us suppose $x \in [\bot, p^\star] \cap [q^\star, \top^\star, p^\star]$. By Lemma 4.2.20 and Definition 4.2.13

$$\Phi^{-1}(x) = \{y; \bot, \top; y\} \subseteq [\{(p; \bot, q; \bot), (\bot, p; \bot)\}] \subseteq [\Psi^{-1}(R)]$$

Similarly if $x \in [q^\star, \top] \cap [q^\star, \top^\star, p^\star]$

Thus for all $x \in [R]$, $\Phi^{-1}(x) \subseteq [\Psi^{-1}(R)]$. This implies

$$\Phi^{-1}([R]) \subseteq [\Psi^{-1}(R)]$$

– Now let us prove $[\Psi^{-1}(R)] \subseteq \Phi^{-1}([R])$.

$$\begin{aligned}
[R] &= [R \setminus \Pi\mathcal{L}_{-1}] && \text{Lemma 4.3.14}\\
&= \bigcup_{r \in R\setminus\Pi\mathcal{L}_{-1}} [r]\\
&= \bigcup_{r \in R\setminus\Pi\mathcal{L}_{-1}, i\in\Psi^{-1}(r)} [\Psi(i)]\\
&= \bigcup_{r \in R\setminus\Pi\mathcal{L}_{-1}, i\in\Psi^{-1}(r)} \Phi([i]) && \text{Proposition 4.2.44}\\
&= \Phi\Big(\bigcup_{r \in R\setminus\Pi\mathcal{L}_{-1}} [\Psi^{-1}(r)]\Big)\\
&= \Phi([\Psi^{-1}(R \setminus \Pi\mathcal{L}_{-1})])\\
[R] &= \Phi([\Psi^{-1}(R)])
\end{aligned}$$

□

4.3.3 Maximal cubical cover as a quotient of the 2-unfolding

In Theorem 4.3.15, we have already proven that conservative regions, and hence the authorized and forbidden region of a program, can be lifted to their unfolding not only point-wise via Φ but also cover-wise via Ψ.

In this part we will extend this lifting property to the normal form of regions. Indeed, as stated in the following Theorem 4.3.27, Ψ^{-1} sends the normal form of a conservative region X to the normal form of its unfolding $\Phi^{-1}(X)$.

Theorem 4.3.27. *Given a conservative program $P \in$ **Pgrm** and R a conservative maximal cover of P, we have*

$$\mathcal{C}^{\max}(\Psi^{-1}(R)) = \Psi^{-1}(R)$$

Theorem 4.3.27 implies, for conservative programs, that the normal form of the forbidden region can be computed as the image by Ψ of the forbidden region of the unfolding, which is loop free.

To prove Theorem 4.3.27, we will first prove that Ψ preserves the orders on cubes (Definition 4.2.28) in Proposition 4.3.19 and that Ψ reflects maximal cubes, up to equivalence, in Proposition 4.3.24.

4.3.3.1 Ψ preserves inclusion of cubes

Before we can prove our result, we prove here two very important technical lemmas. First, Lemma 4.3.16 which stipulates that the equivalence on paths defined in Definition 4.2.13 can be extended to cubes of P^* that do not cross the core of the loop $\top^*$. This implies, that inclusion is obtained inductively by looking at the inclusion of cubes on P.

Then, Lemma 4.3.17 which stipulates that when such cubes are included in a composite cube, they are actually included in one of the two cubes of the composite.

Lemma 4.3.16. *Let $P \in$ **Prcs**, let $p, q, x, y \in \mathcal{P}(P)$ such that $x \leq y$, and $p \leq q$,*

- *If* $(x,y) \neq (\bot,\top)$*, then* $(p,q) \subseteq (x,y) \implies (p^\star,q^\star) \subseteq (x^\star,y^\star)$
- *If* $(x^\star,y^\star)$ *and* $(p^\star,q^\star) \neq (\top^\star,\top^\star)$*, then* $(p^\star,q^\star) \subseteq (x^\star,y^\star) \implies (p,q) \subseteq (x,y)$

Proof. First, let us remark that $(x^\star,y^\star),(p^\star,q^\star) \neq (\top^\star,\top^\star)$ or $(x,y) \neq (\bot,\top)$ implies $p,q \neq \bot,\top$ and $x,y \neq \bot,\top$.

- Let $(p,q) \subseteq (x,y)$ and let $\pi^* \in (p^\star,q^\star)$. Thus, by Definition 4.2.13, $\pi \in (p,q)$, i.e. there exists ν,μ such that $\nu \cdot \pi \cdot \mu \in (x,y)$. Once again by Definition 4.2.13, $(\nu \cdot \pi \cdot \mu)^* = \nu^* \cdot \pi^* \cdot \mu^* \in (x^\star,y^\star)$. Thus, $(p^\star,q^\star) \subseteq (x^\star,y^\star)$
- Let $(p^\star,q^\star) \subseteq (x^\star,y^\star)$ and $\pi \in (p,q)$. By Definition 4.2.13, $\pi^* \in (p^\star,q^\star)$. Such that, there exists ν^*,μ^* such that $\nu^* \cdot \pi^* \cdot \mu^* \in (x^\star,y^\star)$. By Lemma 4.2.14, we can apply Definition 4.2.13, such that $\nu \cdot \pi \cdot \mu \in (x,y)$. Thus, $(p,q) \subseteq (x,y)$. □

Lemma 4.3.17. *Given a program* $P \in \mathbf{Prcs}$ *and positions* $p,q,x,y \in \mathcal{P}(P)$ *such that* $p \leq q$*,* $x,y \notin \{\bot,\top\}$ *we have*

$$(p^\star,q^\star) \subseteq (x^\star,\top^\star,y^\star) \text{ implies } (p^\star,q^\star) \subseteq (x^\star,\top^\star) \text{ or } (p^\star,q^\star) \subseteq (\bot^\star,y^\star)$$

Proof. Let $P \in \mathbf{Prcs}$. Let $p,q,x,y \in \mathcal{P}(P)$ such that $p \leq q$, $x,y \notin \{\bot,\top\}$ and such that $(p^\star,q^\star) \subseteq (x^\star,\top^\star) \times (\bot^\star,y^\star)$.

- If $(p^\star,q^\star) = (\bot^\star,\bot^\star)$, then
$$(p^\star,q^\star) = (\bot^\star,\bot^\star) \subseteq (x^\star,\top^\star)$$
- Otherwise, $(p^\star,q^\star) \neq (\bot^\star,\bot^\star)$. Let $\pi \in (p^\star,q^\star) \subseteq (x^\star,\top^\star) \times (\bot^\star,y^\star)$. This implies there exists $\rho^*,\sigma^* \in (x^\star,\top^\star) \times (\bot^\star,y^\star)$ and $\mu^*,\nu^* \in (x^\star,p^\star) \times (y^\star,q^\star)$ such that
$$\nu^* \cdot \pi^* \cdot \mu^* = \sigma^* \cdot \rho^*$$
By the contraposition of Lemma 4.2.14, $\bot^\star \notin \pi^*$. Thus, $\bot^\star \in [\mu^*] \cup [\nu^*]$. As σ^* and ρ^* are loop free non-empty paths and $\top^\star$ is one of their respective extremities, this implies that $\top^\star$ appears only once in $\sigma^* \cdot \rho^* = \mu^* \cdot \pi^* \cdot \mu^*$. Thus, we have either $\bot^\star \in \mu^*$ or $\bot^\star \in \nu^*$.
  - Let us suppose $\bot^\star \in \mu^*$. Then $\bot^\star \notin \nu^* \cdot \pi^*$. Then by Definition 4.2.13, $\pi,\nu \in (p,q) \times (q,y)$. Which implies for position of a program without loops, $\bot \leq p \leq q \leq y$ i.e. $(p,q) \subseteq (\bot,y)$. Once again by Definition 4.2.13 $(p^\star,q^\star) \subseteq (\bot^\star,y^\star)$
  - Similarly, if we suppose $\bot^\star \in \nu^*$, we can prove $(p^\star,q^\star) \subseteq (x^\star,\top^\star)$. □

We are now ready to prove that Ψ preserves order on cubes (Proposition 4.3.19). This property is first proved on single processes, then extended to parallel composition using the Proposition 4.3.7.

Proposition 4.3.18. *Given a program* $P \in \mathbf{Prcs}$*, then*

$$\begin{aligned} \Psi_{P^\star} \colon \mathcal{C}(\mathcal{U}(P^\star)) \setminus \mathcal{C}_{-1} &\to \mathcal{C}(P^\star) \setminus \mathcal{L}_{-1} \\ c &\mapsto \Psi_{P^\star}(c) \end{aligned}$$

is a surjective morphism of posets, i.e. for all $c,d \in \mathcal{C}(\mathcal{U}(P^\star)) \setminus \mathcal{C}_{-1}$*,*

$$c \subseteq d \implies \Psi(c) \subseteq \Psi(d)$$

Proof. Let us suppose given a program $P \in \mathbf{Prcs}$ and $c, d \in \mathcal{C}(P;P) \setminus \mathcal{C}_{-1}$ such that $c \subseteq d$. Let us prove that $\Psi(c) = \Psi(d)$ by differentiating on the subset of $\mathcal{C}(P;P)$ that c, d belong to.

- $c \in \mathcal{C}_0 = \{(\bot,\bot), (\top,\top), (\bot;\bot,\bot;\bot), (\top;\bot,\top;\bot), (\top;\top,\top;\top)\}$
  - $c = (\bot,\bot)$. Then $c \subseteq d$ implies $\bot \in d$. Hence, $d \in \mathcal{C}_0 \bigcup \mathcal{C}_2 \bigcup \mathcal{C}_1$.
    - $d \in \mathcal{C}_0$. Then $\bot \in d$ implies $d = (\bot,\bot)$. Thus, $\Psi(c) = \Psi(d)$
    - $d \in \mathcal{C}_2$, Then $\Psi(c) = (\bot,\bot) \subseteq (\bot, p^*) = \Psi(d)$.
    - $d \in \mathcal{C}_1$, Then $\Psi(c) \subseteq (\bot,\top) = \Psi(d)$
  - $c = (\bot;\bot,\bot;\bot)$. Similar to the case above.
  - $c = (\top,\top)$. Dual to the case $c = (\bot,\bot)$.
  - $c = (\top;\top,\top;\top)$. Similar to the case above.
  - $c = (\top;\bot,\top;\bot)$. Then $c \subseteq d$ implies $(\top;\bot) \in d$, such that $d \in \mathcal{C}_4 \bigcup \mathcal{C}_4 \bigcup \mathcal{C}_0$.
    - $d = (x;\bot,\top;y) \in \mathcal{C}_4 \bigcup \mathcal{C}_4$. Then by Lemma 4.2.20 $\Psi(c) = (\bot^*,\bot^*) \subseteq (x^*,\top^*) \times (\bot^*, y^*) = \Psi(d)$
    - $d \in \mathcal{C}_0$. Then, $c = d$ and $\Psi(c) = \Psi(d)$.
- $c \in \mathcal{C}_1$ such that $c = (\bot,\top)$. Then $c \subseteq d$ implies $d = (\bot,\top) = c$.
- $c \in \mathcal{C}_2$. Then $c \subseteq d$ implies $\bot \in d$, $d \neq (\bot,\bot)$ such that $d \in \mathcal{C}_2 \bigcup \mathcal{C}_1$.
  - Let $d \in \mathcal{C}_2$. Then $c = (\bot,\bot;p) \subseteq (\bot,\bot;x) = d$ implies

    $$(\bot;\bot,\bot;p) \subseteq (\bot;\bot,\bot;x)$$

    By induction rules this implies $(\bot,p) \subseteq (\bot,x)$. Furthermore, $x \neq \top$ by definition of $\mathcal{C}_2$. Hence,

    $$\begin{array}{ll} \Psi(c) = (\bot,p^*) & \\ \quad = (\bot,\bot^*) \times (\bot^*,p^*) & \text{Lemma 4.2.20} \\ \quad \subseteq (\bot,\bot^*) \times (\bot^*,x^*) & \text{Lemma 4.3.16} \\ \quad = (\bot,y^*) & \text{Lemma 4.2.20} \\ \Psi(c) \subseteq \Psi(d) & \end{array}$$

  - Let $d \in \mathcal{C}_1$. Then $\Psi(c) \subseteq (\bot,\top) = \Psi(d)$
- $c \in \mathcal{C}_3$. Dual of the case $c \in \mathcal{C}_2$
- $c \in \mathcal{C}_4$, $c = (p;\bot,\top;q) = (p;\bot,\top;\bot) \times (\top;\bot,\top;q)$. Thus, $\top;\bot \in c$. Then $c \subseteq d$ implies $\top;\bot \in d$ and thus $d \in \mathcal{C}_4 \bigcup \mathcal{C}_1$.
  - Let $d \in \mathcal{C}_1$. Then $\Psi(c) \subseteq (\bot,\top) = \Psi(d)$
  - Let $d \in \mathcal{C}_4$. Then $d = (x;\bot,\top;y) = (x;\bot,\top;\bot) \times (\top;\bot,\top;y)$. Furthermore, $c \subseteq d$ implies

    $$(p;\bot,\top;\bot) \times (\top;\bot,\top;q) \subseteq (x;\bot,\top;\bot) \times (\top;\bot,\top;y)$$

By induction rules, we get

$$(p, \top) \subseteq (x, \top) \text{ and } (\bot, q) \subseteq (\bot, y)$$

By Proposition 4.2.33, $x \neq \bot$ and $y \neq \top$. This implies

$$\begin{array}{ll} \Psi(c) = (p^\star, \top^\star, q^\star) & \text{Lemma 4.2.20} \\ \quad = (p^\star, \top^\star) \times (\bot^\star, q^\star) & \\ \quad \subseteq (x^\star, \top^\star) \times (\bot^\star, y^\star) & \text{Lemma 4.3.16} \\ \quad \subseteq (x^\star, \top^\star, y^\star) & \\ \Psi(c) \subseteq \Psi(d) & \text{Lemma 4.2.20} \end{array}$$

- $c \in \mathcal{C}_5$. Then $c \subseteq d$ implies $d \bigcap \{x\,;\bot \mid x \in \mathcal{P}(P) \setminus \{\top\}\} \neq \emptyset$. Thus, $d \notin \mathcal{C}_0 \bigcup \mathcal{C}_3 \bigcup \mathcal{C}_6$
  - $d \in \mathcal{C}_0$. Then $c \subseteq d$ implies $c = d$.
  - $d \in \mathcal{C}_1$. Then $\Psi(c) \subseteq (\bot, \top) = \Psi(d)$
  - $d \in \mathcal{C}_2$. Then $c = (p\,;\bot, q\,;\bot) \subseteq (\bot\,;\bot, x, \bot\,;) \subseteq (\bot, x\,;\bot)$. By induction rules $(p, q) \subseteq (\bot, x)$. Furthermore, by definition of $\mathcal{C}_2$, $x \neq \top$. Thus, by Lemma 4.3.16, $\Psi(c) = (p^\star, q^\star) \subseteq (\bot^\star, x^\star) \subseteq (\bot, x^\star) = \Psi(d)$.
  - $d \in \mathcal{C}_4$. Then $c = (p\,;\bot, q\,;\bot) \subseteq (p\,;\bot, \top\,;\bot) \subseteq (x\,;\bot, \top\,;\bot) \subseteq (x\,;\bot, \top\,;y) = d$. By induction rules this implies $(p, q) \subseteq (p, \top) \subseteq (x, \top)$. Furthermore, $x \neq \bot$ by definition of $\mathcal{C}_4$. Hence,

$$\begin{array}{ll} \Psi(c) = (p^\star, q^\star) & \\ \quad \subseteq (x^\star, \top^\star) & \text{Lemma 4.3.16} \\ \quad \subseteq (x^\star, \top^\star) \times (\bot^\star, y^\star) & \\ \quad \subseteq (x^\star, \top^\star, y^\star) & \\ \Psi(c) \subseteq \Psi(d) & \text{Lemma 4.2.20} \end{array}$$

  - $d \in \mathcal{C}_5$. Then $c = (p\,;\bot, q\,;\bot) \subseteq (x\,;\bot, y, \bot\,;) = d$ implies by induction rules $(p, q) \subseteq (x, y) \neq (\bot, \top)$. Thus, by Lemma 4.3.16, $\Psi(c) = (p^\star, q^\star) \subseteq (x^\star, y^\star) = \Psi(d)$.
- $c \in \mathcal{C}_6$. Similar to the case $c \in \mathcal{C}_5$. □

Proposition 4.3.19. *Given a program $P \in \mathbf{Pgrm}$, we have that*

$$\begin{aligned} \Psi \colon \mathcal{C}(\mathcal{U}(P)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} &\to \mathcal{C}(P) \setminus \Pi_P \mathcal{L}_{-1} \\ c &\mapsto \Psi(c) \end{aligned}$$

is a surjective morphism of posets.

Proof. By induction on P

- $P \in \mathbf{Prcs}$. Then Ψ is the identity.
- $P = Q^\star$. By Proposition 4.3.18 and Proposition 4.2.36.

- $P = S \| T$. Let $s \times t \subseteq x \times y \in \mathcal{C}^m_{\mathcal{U}(P)} \setminus \Phi^{-1}(R)$. Then $s \subseteq x$ and $t \subseteq y$ by Definition 4.2.21. By induction hypothesis, this implies, $\Psi(s) \subseteq \Psi(x)$ and $\Psi(t) \subseteq \Psi(y)$. Then once again, by Definition 4.2.21

$$\Psi(s \times t) \subseteq \Psi(x \times y)$$

As $\Psi_{S\|T} = \Psi_S \times \Psi_T$, by induction hypothesis, $\Psi_{S\|T}$ is surjective.

□

4.3.3.2 Ψ weakly reflects inclusion of maximal cubes

Before we prove our result, once again, we first prove a few characteristics of maximal cubes of loops. Indeed, for a program $P^\star \in \mathbf{Prcs}^*$ the subsets $\mathcal{C}_0$ and $\mathcal{L}_0$ of $\mathcal{C}(P^\star)$ are never maximal. Furthermore, using Lemma 4.3.12 and the aforementioned properties, we then get a characterization of any maximal cover R of $P^\star$ in Proposition 4.3.21. Finally, we will prove that Ψ weakly reflects inclusion of maximal cubes (i.e. up to the equivalence ofDefinition 4.2.37)

Lemma 4.3.20. *Given a program $P \in \mathbf{Pgrm}$ and R a maximal conservative cover on $\mathcal{P}(P)$, we have*

$$R \bigcap \Pi_P \mathcal{L}_0 = \emptyset$$

Proof. Let $P \in \mathbf{Pgrm}$ and R a maximal conservative cover of P. If $[R] = \mathcal{P}(P)$, then $R = \{(\bot, \top)\}$, which concludes the proof. We can then suppose that $[R] \neq \mathcal{P}(P)$ for the rest of the proof.

- $P \in \mathbf{Prcs}$. Then $\Pi_P \mathcal{L}_0 = \emptyset$ by definition
- $P = P^\star$. Then $\Pi_P \mathcal{L}_0 = \{(\bot, \bot), (\top, \top), (\top^\star, \top^\star)\}$.
  - If $\bot^\star \notin [R]$. Then by conservativity of R, this implies $\bot, \top, \bot^\star \notin [R]$, i.e.

    $$[\bot, \bot] \cap [R] = \emptyset \text{ and } [\top^\star, \top^\star] \cap [R] = \emptyset \text{ and } [\top, \top] \cap [R] = \emptyset$$

    This implies $R \bigcap \Pi_P \mathcal{L}_0 = \emptyset$.
  - If $\bot^\star \in [R]$, then by Lemma 4.3.12, $(\bot, \top^\star, \top) \in R$, furthermore by definition of composite cubes,

    $$(\bot, \bot) \subseteq (\bot, \top^\star, \top) \text{ and } (\top^\star, \top^\star) \subseteq (\bot, \top^\star, \top) \text{ and } (\top, \top) \subseteq (\bot, \top^\star, \top)$$

    Thus no cubes of $\Pi_P \mathcal{L}_0$ are in R as they are not maximal.
- $P = P_1 \| P_2$. By Proposition 4.3.7, for all $c = c_1 \times c_2 \in R$ the sets

  $$\pi_1 R_c = \{c' \in \mathcal{P}(P_1) \mid c' \times c_2 \in R\} \text{ and } \pi_1 R_c = \{c' \in \mathcal{P}(P_2) \mid c_1 \times c' \in R\}$$

  are conservative and maximal. By definition, $c_i \in \pi_i R_c$ for $i = 1, 2$ and by induction hypothesis, this implies $c_i \notin \Pi_{P_i} \mathcal{L}_0$. Thus, $c \notin \Pi_P \mathcal{L}_0$

□

Proposition 4.3.21. *Given a program $P \in \mathbf{Prcs}$ and R a conservative maximal region on the program $P^\star$, we have the three mutually exclusive characterization of R:*

- *If $[R] = \mathcal{P}(P^\star)$, then $R = \{(\bot, \top)\}$*
- *If $[R] \neq \mathcal{P}(P^\star)$ and $\bot^\star \in [R]$, then there exists $((p_j, q_j))_{j \in J_1 \cup J_2}$, such that for all $j \in J_1 \cup J_2$, we have $p_j, q_j \in \mathcal{P}(P) \setminus \{\bot, \top\}$ and*

$$R = \bigcup_{j \in J_1} \{(p_j^\star, q_j^\star)\} \bigcup_{j \in J_2} \{(\bot, q_j^\star), (p_j^\star, \top), (p_j^\star, \top^\star, q_j^\star), (\bot, \top^\star, \top)\}$$

- *Otherwise, there exists $((p_j, q_j))_{j \in J} \in \mathcal{C}(P)^J$, such that for all $j \in J$, $p_j, q_j \notin \{\bot, \top\}$ and*

$$R = \bigcup_{j \in J} \{(p_j^\star, q_j^\star)\}$$

Proof. Let us suppose R a conservative maximal region on a program $P^\star \in \mathbf{Prcs}^*$.

- If $[R] = \mathcal{P}(P^\star)$, then $(\bot, \top) \in \mathcal{C}([R])$ and for any cube $c \in \mathcal{C}(P^\star)$, $c \subseteq (\bot, \top)$, thus $R = \{\bot, \top\}$.
- If $[R] \neq \mathcal{P}(P^\star)$ and $\bot^\star \in [R]$. By Lemma 4.3.20 and Lemma 4.3.20,

$$R \subseteq \mathcal{L}_{-1} \cup \mathcal{L}_2 \cup \mathcal{L}_3 \cup (\mathcal{L}_4 \cup \mathcal{L}_4) \cup \mathcal{L}_5$$

  Then, by Lemma 4.3.12, for every pair of positions $p, q \neq \bot, \top$ such that $(p^\star, \top)$,$(\bot, q^\star) \in R$ or $(p^\star, \top^\star, q^\star) \in R$ we have that

$$(p^\star, \top), (\bot, q^\star), (p^\star, \top^\star, q^\star), (\bot, \top^\star, \top) \in R$$

  Furthermore as R is conservative, we can exclude the case where only $(p^\star, \top)$ or $(\bot, q^\star)$ would belong to R. Indeed, if we define

$$X_\bot = \{x \in \mathcal{P}(P) \mid \bot < x < \top, (\bot, x^\star) \in \mathcal{C}(R)\}$$
$$Y_\top = \{x \in \mathcal{P}(P) \mid \bot < x < \top, (y^\star, \top) \in \mathcal{C}(R)\}$$

  Let us take p (resp. q) the maximal (resp. minimal) elements of $X_\bot$, (resp. $Y_\top$). We have seen in the proof of Lemma 4.3.12 that these are correctly defined and

$$(\bot, p^\star), (q^\star, \top) \in R$$

  Now we only need to prove that the cubes of $R \cap \mathcal{L}_5$ do not have $\top^\star$ as an endpoint. By contradiction, let us suppose that there exists such a cube $(\bot^\star, q^\star) \in R$. Then by definition

$$(\bot^\star, q^\star) \subseteq (\bot, q^\star) \in \mathcal{C}([R])$$

  This contradicts the definition of R as maximal. Thus, $(\bot^\star, q^\star) \notin R$ (the case $(p^\star, \top^\star)$ is dual).
- Otherwise, the only possible maximal cubes of R are cubes of $\mathcal{L}_5$ such that the support does not intersect $\top^\star$. By Lemma 4.2.14, this is precisely the cubes of the form $(p^\star, q^\star)$ such that $p, q \neq \bot, \top$. □

Proposition 4.3.22. *Given a program $P \in \mathbf{Prcs}$ such that $P \neq \mathtt{P}_a, \mathtt{V}_a$, a maximal conservative cover R of $P^\star$ and $c, d \in \mathcal{C}^{\max}(\Psi^{-1}(R))$, we have*

$$\Psi(c) \subseteq \Psi(d) \implies \exists k \in \Psi^{-1}(\Psi(d)), i \subseteq k$$

Proof. Let us suppose given a program $P \in \mathbf{Prcs}$, a maximal conservative cover R of $P^\star$ and $c, d \in \mathcal{C}^{\max}(\Psi^{-1}(R))$. Furthermore, let us suppose $\Psi(c) \subseteq \Psi(d)$. Let us prove that

$$\exists k \in \Psi^{-1}(\Psi(d)), i \subseteq k$$

First let us remark that by definition of Ψ , and by Lemma 4.3.20

$$\Psi(c), \Psi(d) \notin \mathcal{L}_{-1} \cup \mathcal{L}_0$$

We will now proceed to check all cases depending on the subset of $\mathcal{C}(P^\star)$ in which c, d are contained.

- $\Psi(d) \in \mathcal{L}_1$. Then by definition for all $c \in \mathcal{C}(\Phi^{-1}(R))$, $c \subseteq (\bot, \top) = d$.
- $\Psi(d) \in \mathcal{L}_2$, i.e. $\Psi(d) = (\bot, y^\star)$. Then $\Psi(c) \subseteq \Psi(d)$ implies by looking at the supports: $\Psi(c) \notin \mathcal{L}_3 \bigcup \mathcal{L}_1$.
  - $\Psi(c) \in \mathcal{L}_2$. Then $\Psi(c) = (\bot, q^\star) \subseteq (\bot, y^\star) = \Psi(d)$ implies by Lemma 4.2.20 $(\bot^\star, q^\star) \subseteq (\bot^\star, y^\star)$. Furthermore, we have $y, q \neq \top$. Then
    - $y = \bot$ implies $q = \bot$. Thus, $c = d$
    - $q = \bot$ implies $c = (\bot, \bot\,;\bot) \subseteq (\bot, y\,;\bot) = d$.
    - Otherwise, $y, q \notin \{\bot, \top\}$ Thus by Lemma 4.3.16, $(\bot, q) \subseteq (\bot, y)$ and by induction rules $c = (\bot, q\,;\bot) \subseteq (\bot, y\,;\bot) = d$
  - $\Psi(c) \in \mathcal{L}_4$, i.e. $\Psi(c) = (p^\star, \top^\star, q^\star)$, $p, q \notin \{\bot, \top\}$. Let us prove by contradiction that
    $$\Psi(c) \not\subseteq \Psi(d)$$
    Let us suppose, $\Psi(c) \subseteq \Psi(d)$, then for any $\sigma \cdot \rho \in \Psi(c)$, there exists $\mu, \nu \in (\bot, p^\star) \times (q^\star, y^\star)$ such that
    $$\nu \cdot \sigma \cdot \rho \cdot \mu \in (\bot, y^\star)$$
    By definition, $\top^\star \in \sigma \cdot \rho$ and $\top^\star \in \mu$. Furthermore, as $\top^\star$ is not the endpoint of μ as $p \neq \bot, \top$. Thus, there is a non-trivial loop in $\nu \cdot \sigma \cdot \rho \cdot \mu$ contradicts the fact that it is an element of $(\bot, y^\star)$.
  - $\Psi(c) \in \mathcal{L}_5$. $\Psi(c) = (p^\star, q^\star) \neq (\bot^\star, \bot^\star)$. Then $\Psi(c) = (p^\star, q^\star) \subseteq (\bot^\star, y^\star) = \Psi(d)$ implies $y \neq \bot$. As $y \neq \top$ by definition, this implies by Lemma 4.3.16, $(p, q) \subseteq (\bot, y)$.
    - If $c \in \mathcal{C}_5$, this implies $c = (p\,;\bot, q\,;\bot) \subseteq (\bot\,;\bot, y\,;\bot) \subseteq (\bot, y\,;\bot) = d$.
    - Otherwise, $c \in \mathcal{C}_6$. Then $(p, q) \subseteq (\bot, y)$
      $$c = (\top\,;\bot, \top\,;q) \in \mathcal{C}^{\max}(\Psi^{-1}(R))$$
      As $P \neq \mathtt{P}_a, \mathtt{V}_a$, by Definition 4.3.2, there exists $z \in \mathcal{P}(P)$, such that $z\,;\bot \to \top\,;\bot$ is pure. As R is conservative, so is $\Psi^{-1}(R)$. This implies that
      $$c \subset (z\,;\bot, \top\,;q) \in \mathcal{C}([\Psi^{-1}(R)])$$
      which contradicts the fact that c is maximal.

- $\Psi(d) \in \mathcal{L}_3$. Symmetric to the case above.
- $\Psi(d) \in \mathcal{L}_4$, such that $\Psi(d) = (x^\star, \top^\star, y^\star)$ and $x, y \notin \{\bot, \top\}$. Then $\Psi(c) \subseteq \Psi(d)$ implies $\Psi(c) \in \mathcal{L}_4 \bigcup \mathcal{L}_5$.
  - $\Psi(c) \in \mathcal{L}_4$. $\Psi(c) = (p^\star, \top^\star, q^\star)$, $p, q \notin \{\bot, \top\}$. Then $\Psi(c) \subseteq \Psi(d)$ implies $(p^\star, \top^\star) \times (\top^\star, q^\star) \subseteq (x^\star, \top^\star) \times (\top^\star, y^\star)$ i.e.
  $$(p^\star, \top^\star) \subseteq (x^\star, \top^\star) \text{ and } (\top^\star, q^\star) \subseteq (\top^\star, y^\star)$$
  By Lemma 4.3.16, this implies $(p, \top) \subseteq (x, \top)$ and $(\bot, q) \subseteq (\bot, y)$. Thus, by induction rules $c = (p; \bot, \top; q) \subseteq (x; \bot, \top; y) = d$
  - $\Psi(c) \in \mathcal{L}_5$. $\Psi(c) = (p^\star, q^\star) \subseteq (x^\star, \top^\star, y^\star) = \Psi(d)$ implies by Lemma 4.3.17,
  $$(p^\star, q^\star) \subseteq (x^\star, \top^\star) \text{ or } (p^\star, q^\star) \subseteq (\bot^\star, y^\star)$$
  Let us suppose $(p^\star, q^\star) \subseteq (x^\star, \top^\star)$, and $c \in \mathcal{C}_5$. By definition $(p^\star, q^\star)$ and $(\bot^\star, y^\star) \neq (\top^\star, \top^\star)$. Thus, by Lemma 4.3.16, $(p, q) \subseteq (x, \top)$ and by induction rules,
  $$c = (p; \bot, q; \bot) \subseteq (x; \bot, \top; \bot) \subseteq x; \bot, \top; y = d$$
  The cases where $(p^\star, q^\star) \subseteq (y^\star, \top^\star)$ and/or $c \in \mathcal{C}_6$ follow the same proof.
- $\Psi(d) \in \mathcal{L}_5$. $\Psi(d) = (x^\star, y^\star) \neq (\bot^\star, \bot^\star)$, $p \leq q$. Then $\Psi(c) \subseteq \Psi(d)$ implies by looking at the supports that $\Psi(c)$ can only belong to $\mathcal{L}_4 \bigcup \mathcal{L}_5$
  - $\Psi(c) \in \mathcal{L}_4$, i.e. $\Psi(c) = (p^\star, \top^\star, q^\star)$, $p, q \notin \{\bot, \top\}$. Let us prove by contradiction that
  $$\Psi(c) \not\subseteq \Psi(d)$$
  Let us suppose $\Psi(c) \subseteq \Psi(d)$, then for any $\sigma \cdot \rho \in \Psi(c)$ there exists $\mu, \nu \in (x^\star, p^\star) \times (q^\star, y^\star)$ such that
  $$\nu \cdot \sigma \cdot \rho \cdot \mu \in (x^\star, y^\star)$$
  The contraposition of Lemma 4.2.14 implies $\top \notin \nu \cdot \sigma \cdot \rho \cdot \mu$, which contradicts the fact that by definition $\top^\star \in \sigma \cdot \rho$.
  - $\Psi(c) \in \mathcal{L}_5$. $\Psi(c) = (p^\star, q^\star) \neq (\bot^\star, \bot^\star)$. Then $\Psi(c) = (p^\star, q^\star) \subseteq (x^\star, y^\star) = \Psi(d)$ implies by Lemma 4.3.16, $(p, q) \subseteq (x, y)$ and by induction rules
  $$(p; \bot, q; \bot) \subseteq (x; \bot, y; \bot) \approx d$$
  $$(\top; p, \top; q) \subseteq (\top; x, \top; y) \approx d$$
  As $c \in \{(p; \bot, q; \bot), (\top; p, \top; q)\}$ by definition, this concludes the proof. □

Remark 4.3.23. If the program $P^\star$ is conservative, then $P \neq \mathtt{P}_a, \mathtt{V}_a$, as otherwise $\Delta(P) \neq 0$. Thus, for all $P \in \mathbf{Pgrm}$, with P conservative there are no sub-programs of the form $\mathtt{P}_a^\star$ or $\mathtt{V}_a^\star$.

Proposition 4.3.24. *Given a conservative program $P =\in \mathbf{Pgrm}$, a maximal conservative cover R of P and $c, d \in \mathcal{C}^{\max}(\Psi^{-1}(R))$, we have*

$$\Psi(c) \subseteq \Psi(d) \implies \exists d' \in \Psi^{-1}(\Psi(d)) \text{ such that } c \subseteq d'$$

Proof. By induction on P.

- $P \in \mathbf{Prcs}$. Trivial.
- $P \in \mathbf{Prcs}^*$. By Remark 4.3.23 we can apply Proposition 4.3.22.
- $P = P_1 \parallel P_2$. Let $c, d \in \mathcal{C}(\Psi^{-1}(R))$, i.e. $c = c_1 \times c_2$, $d = d_1 \times d_2$. Then, by Proposition 4.3.7

$$c = c_1 \times c_2 \in \pi_1 R_c \times \pi_2 R_c$$
$$d = d_1 \times d_2 \in \pi_1 R_d \times \pi_2 R_d$$

with $\pi_i R_c$ and $\pi_i R_d$ conservative maximal. Then $\Psi(c) \subseteq \Psi(d)$ implies

$$\Psi(c_1) \subseteq \Psi(d_1) \text{ and } \Psi(c_2) \subseteq \Psi(d_2)$$

By induction hypothesis, there exists $d' = d'_1 \times d'_2 \in \Psi^{-1}(\Psi(d_1)) \times \Psi^{-1}(\Psi(d_2)) = \Psi^{-1}(\Psi(d))$ such that $c \subseteq d'$. □

4.3.3.3 Maximal cubical cover as a quotient of the 2-unfolding

Given a conservative program P, Ψ_P actually lifts all cubes of a conservative region X to the cubes of its unfolding $\Phi^{-1}(X)$. But this lifting is not surjective, as all cubes in $\Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$ do not have an image by Ψ. This constraint disappears when considering the maximal cubes, as cubes of $\Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$ are not maximal in unfolding of conservative regions (proved in Proposition 4.3.25 below).

One thing to remember is that the cube $(\bot, \top^*, \top)$ (and thus all cubes in $\Pi_P \mathcal{L}_{-1}$) do not have an antecedent by Ψ.

Proposition 4.3.25. *Given a program $P \in \mathbf{Pgrm}$ and R a maximal conservative cover on $\mathcal{P}(P)$, we have*

$$\mathcal{C}^{\max}([\Psi^{-1}(R)]) \bigcap \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} = \emptyset$$

Proof. Let $P \in \mathbf{Pgrm}$ and R a maximal conservative cover of P. If $[R] = \mathcal{P}(P)$, then $R = \{(\bot, \top)\}$ by Proposition 4.3.21, which concludes the proof. We can then suppose that $[R] \neq \mathcal{P}(P)$ for the rest of the proof. Let us prove the property by induction on the program P.

- $P \in \mathbf{Prcs}$. Then $\Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} = \emptyset$, which concludes the proof.
- $P = Q^*$. Then $\mathcal{C}^m_{\mathcal{U}(P)} \setminus \Psi^{-1}(R) = \mathcal{C}^m_{Q;Q} \setminus \Phi^{-1}(R)$.
  - If $[R] = \mathcal{P}(P)$, then $R = \{(\bot, \top)\}$ and thus
  $$\Psi^{-1}(R) = \{(\bot, \top)\} = \mathcal{C}^{\max}([\Psi^{-1}(R)])$$
  - Otherwise, let us prove by contradiction that
  $$\mathcal{C}^{\max}([\Psi^{-1}(R)]) \bigcap \mathcal{C}_{-1} = \emptyset$$

Let us suppose $c \in \mathcal{C}_{-1} \cap \mathcal{C}^{\max}(\Phi^{-1}[R])$. This implies

$$[\bot\,;\bot, \top\,;\bot] \subseteq \Phi^{-1}[R] \text{ or } [\top\,;\bot, \top\,;\top] \subseteq \Phi^{-1}[R]$$

In both cases, this implies $\{q^\star \mid q \in \mathcal{P}(Q)\} \subseteq \Phi(\Phi^{-1}[R]) = [R]$. By conservativity $\top^* \in R$ also implies $\bot, \top \in R$. Thus, $\mathcal{P}(Q^\star) \subseteq R$, which contradicts our hypothesis on R. Thus,

$$\mathcal{C}_{-1} \cap \mathcal{C}^{\max}(\Phi^{-1}[R]) = \emptyset$$

- $P = P_1 \parallel P_2$. By Proposition 4.3.5, $\Phi^{-1}[R]$ is conservative, thus by Theorem 4.3.15, $S = \mathcal{C}^{\max}(\Psi^{-1}(R))$ is also conservative. By Proposition 4.3.7, for all $c = c_1 \times c_2 \in S$ the sets

$$\pi_1 S_c = \{c' \in \mathcal{P}(P_1) \mid c' \times c_2 \in S\} \text{ and } \pi_1 S_c = \{c' \in \mathcal{P}(P_2) \mid c_1 \times c' \in S\}$$

are conservative and maximal. By definition, $c_i \in \pi_i R_c$ for $i = 1, 2$ and by induction hypothesis, this implies $c_i \notin \Pi_{\mathcal{U}(P_i)}\mathcal{C}_{-1}$. Thus, $c \notin \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$

□

Proposition 4.3.26. *Given a conservative program $P \in$ **Pgrm** and R a conservative maximal cover of P, we have*

$$\mathcal{C}(\Psi^{-1}(R)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} = \Psi^{-1}(\mathcal{C}(R))$$

Proof. We recall that $\Psi^{-1}(\mathcal{C}(R)) = \Psi^{-1}\big((\mathcal{C}(R)) \setminus \Pi_P \mathcal{L}_{-1}\big)$. If $[R] = \emptyset$, then

$$\mathcal{C}(\Psi^{-1}(R)) = \emptyset = \Psi^{-1}(\mathcal{C}(R))$$

Which concludes the proof. Thus, for the rest of the proof, we will suppose $[R] \neq \emptyset$.

- Let $c \in \Psi^{-1}(R)$. We want to prove that $c \in \mathcal{C}(\Psi^{-1}(R)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$. By definition of Ψ, $c \notin \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$. Thus, we simply need to prove that

$$c \in \mathcal{C}(\Psi^{-1}(R))$$

Then $c \in \Psi^{-1}(R)$ implies that there exists $d \in R$ such that $\Psi(c) = d$. Then,

$$\begin{aligned}
\Phi([c] \cap [\Psi^{-1}](R)) &\subseteq \Phi[c] \cap \Phi[\Psi^{-1}(R)] \\
&\subseteq [\Psi(c)] \cap \Phi[\Psi^{-1}(R)] && \text{Proposition 4.2.44} \\
&\subseteq [\Psi(c)] \cap \Phi(\Phi^{-1}[R]) && \text{Theorem 4.3.15} \\
&\subseteq [d] \cap [R] \\
\Phi([c] \cap [\Psi^{-1}](R)) &\subseteq \emptyset
\end{aligned}$$

Thus by definition of Φ, this implies $[c] \cap [\Psi^{-1}](R) = \emptyset$ i.e.

$$c \in \mathcal{C}(\Psi^{-1}(R))$$

- Let us prove that
$$\mathcal{C}(\Psi^{-1}(R)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} \subseteq \Psi^{-1}(\mathcal{C}(R))$$
i.e. for all $c \in \mathcal{C}(\Psi^{-1}(R)) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$, we have
$$\Psi(c) \in \mathcal{C}(R)$$
As $c \notin \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}$, $\Psi(c)$ is correctly defined. Now, by contradiction let us suppose there exists $q \in [\Psi(c)] \cap [R]$. $q \in [\Psi(c)] = \Phi[c]$ (by Proposition 4.2.44) implies that there exists $p \in [c]$ such that $\Phi(p) = q$, i.e.
$$p \in \Phi^{-1}(q) \subseteq \Phi^{-1}[R] = [\Psi^{-1}(R)] \quad \textit{Theorem } 4.3.15$$
Then $p \in [c] \cap [\Psi^{-1}(R)] = \emptyset$ by definition. As this is impossible, we get $[\Psi(c)] \cap [R] = \emptyset$, i.e.
$$\Psi((\mathcal{C}(\Psi^{-1}(R))) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1}) \subseteq \mathcal{C}(R) \setminus \Pi_P\mathcal{L}_{-1}$$
Thus by applying Ψ^{-1}, we get
$$(\mathcal{C}(\Psi^{-1}(R))) \setminus \Pi_{\mathcal{U}(P)}\mathcal{C}_{-1} \subseteq \Psi^{-1}(\mathcal{C}(R)) \qquad \square$$

Combining both Proposition 4.3.24 and Proposition 4.3.18, we can finally prove that Ψ^{-1} sends maximal covers onto maximal covers. Thus, Ψ defines an equivalence between the maximal cubes of a conservative region X of a program P and the quotient of the set of maximal cubes of $\Phi^{-1}(X)$ by the equivalence relation of Definition 4.2.37.

Theorem 4.3.27. *Given a conservative program $P \in$ **Pgrm** and R a conservative maximal cover of P, we have*
$$\mathcal{C}^{\max}(\Psi^{-1}(R)) = \Psi^{-1}(R)$$

Proof. We first remind ourselves that, by definition $\Psi^{-1}(R) = \Psi^{-1}(R \setminus \Pi_P\mathcal{L}_{-1})$. Thus, we can instead prove
$$\mathcal{C}^{\max}(\Psi^{-1}(R)) = \Psi^{-1}(R \setminus \Pi_P\mathcal{L}_{-1})$$

- Let us prove
$$\Psi^{-1}(R \setminus \Pi_P\mathcal{L}_{-1}) \subseteq \mathcal{C}^{\max}(\Psi^{-1}(R))$$
Let $c \in R \setminus \Pi_P\mathcal{L}_{-1}, d \in \Psi^{-1}(c)$. By Proposition 4.3.26, $d \in \mathcal{C}(\Psi^{-1}(R))$. Now let us prove that d is maximal. Let us suppose that there exists
$$d' \in \mathcal{C}^{\max}(\Psi^{-1}(R)), d \subseteq d'$$
Then by Proposition 4.3.19
$$c = \Psi(d) \subseteq \Psi(d') \in R \setminus \Pi_P\mathcal{L}_{-1}$$
By maximality of c this implies, $d' \in \Psi^{-1}(c)$. Thus, by the contraposition of Proposition 4.2.40 for $d, d' \in \Psi^{-1}(c)$, we get $d = d'$, thus d is maximal and
$$\Psi^{-1}(R \setminus \Pi_P\mathcal{L}_{-1}) \subseteq \mathcal{C}^{\max}(\Psi^{-1}(R))$$

- We want to show that

$$\mathcal{C}^{\max}(\Psi^{-1}(R)) \subseteq \Psi^{-1}(R \setminus \Pi_P \mathcal{L}_{-1})$$

i.e. that for every $c \in \mathcal{C}^{\max}(\Psi^{-1}(R))$, $\Psi(c) \in R$. By Proposition 4.3.19, $\Psi(c) \in \mathcal{C}(R)$, with R maximal. Thus, there exists $d \in R$, such that

$$\Psi(c) \subseteq d$$

By the inclusion $\Psi^{-1}(R \setminus \Pi_P \mathcal{L}_{-1}) \subseteq \mathcal{C}^{\max}(\Psi^{-1}(R))$ proved above, there exists $c' \in \mathcal{C}^{\max}(\Psi^{-1}(R))$ such that $d = \Psi(c')$, which implies

$$\Psi(c) \subseteq \Psi(c')$$

As $c \in \mathcal{C}^{\max}(\Psi^{-1}(R))$, we can apply Proposition 4.3.24, thus $\exists r \in \Psi^{-1}(\Psi(c')), c \subseteq r$. By maximality of c, $r = c$, i.e. $\Psi(c) = \Psi(r) = \Psi(c') \in R$. □

For programs, in particular for Algorithm 3.3.23, we are given a cover of the forbidden region, and we wish to compute the authorized region from there. This means that we require a bit more than Theorem 4.3.27 if we wish to implement Algorithm 3.3.23 via the unfolding.

Theorem 4.3.28. *Given a conservative program $P \in$ **Pgrm** and R a conservative maximal cover of P, we have*

$$(\Psi^{-1}(R))^{\mathrm{c}} = \Psi^{-1}(R^{\mathrm{c}})$$

where for any cover S, $S^{\mathrm{c}} = \mathcal{C}^{\max}([S]^{\mathrm{c}})$.

Proof. As R is conservative, by Proposition 4.3.6, so is R^{c}. Then by applying Theorem 4.3.27 to R^c maximal conservative, we get

$$\begin{aligned}
\Psi^{-1}(R^{\mathrm{c}}) &= \mathcal{C}^{\max}(\Psi^{-1}(R^{\mathrm{c}})) \\
&= \mathcal{C}^{\max}(\Phi^{-1}([R^{\mathrm{c}}])) && \text{Theorem 4.3.15} \\
&= \mathcal{C}^{\max}(\Phi^{-1}([R]^{\mathrm{c}})) \\
&= \mathcal{C}^{\max}((\Phi^{-1}[R])^{\mathrm{c}}) \\
&= \mathcal{C}^{\max}([\Psi^{-1}(R)^{\mathrm{c}}]) && \text{Theorem 4.3.15} \\
\Psi^{-1}(R^{\mathrm{c}}) &= (\Psi^{-1}(R))^{\mathrm{c}} && \text{by maximality of } (\Psi^{-1}(R))^{\mathrm{c}}
\end{aligned}$$

□

4.4 Deadlock detection algorithm for looped programs

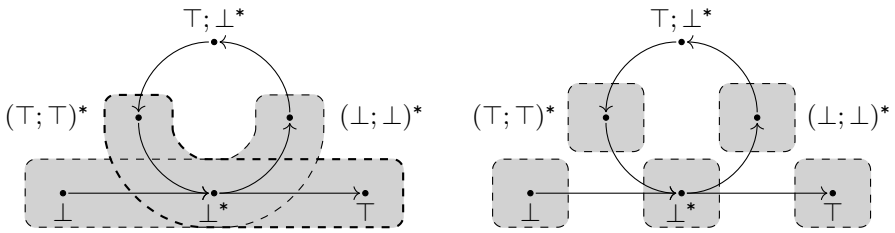
In this section we finally implement the extension of Algorithm 3.3.23 for programs with loops. We recall that the idea behind this algorithm is to first generate a cubical partition of the authorized region that is compatible with the maximal cubical cover. Then, cubes are arranged in order to reflect reachability but on the level of cubes instead of points.

This guarantees that the cubes that do not contain the terminal position $\top$ but do not lead to another cube are deadlocks, Any cube that lead to a deadlock are unsafe, and cube leading only to deadlocks are doomed.

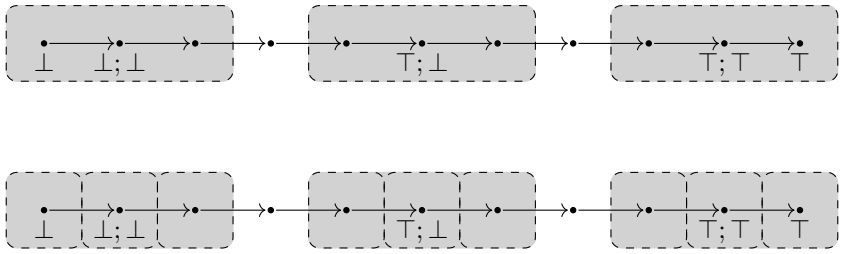
Now that cubes are properly defined on semantics of programs with loops, one could implement intersection and complement of cubes for looped syntactic semantics. There are some difficulties in setting up intersection and complements as they do not behave very well in the case of loops, but it is entirely possible. Proving that the generic partition is still a partition is also not trivial.

We do not opt for this approach, instead, we study the generic partition of the base programs through the generic partition of its unfolding. Indeed, we already know that the maximal covers of forbidden and authorized region lifts, so it is natural to check if the generic partition also lifts directly to its unfolding. This is not the case, but through some manipulation, which consists of separating the cubes at the lifting of the core of the loop, as detailed in Example 4.4.1.

Example 4.4.1. Given a resource of capacity $\kappa_a = 0$, let us consider a very simple program $P = (\mathtt{P}_a\,;\mathtt{V}_a)^*$, whose semantics is given below.



Now if we consider the lifting, on its unfolding $\mathcal{U}(P) = (\mathtt{P}_a\,;\mathtt{V}_a)\,;(\mathtt{P}_a\,;\mathtt{V}_a)$, of the generic partition (on the bottom below) we can see that it is finer than the partition of the lifting (on top below)



Remark 4.4.2. The same phenomena appear for more classical parallel programs. Indeed, if one looks at $P = (\mathtt{P}_a\,;\mathtt{V}_a)^* \,\|\, \mathtt{P}_a;\mathtt{V}_a$, with $\kappa_a = 1$ which would be a "looped" version of the floating square (Example 1.4.3), a similar result holds.

As this separation does not impact reachability of cubes, this transformation does not affect the algorithm. Then to get the generic partition of the base program, as well as the unsafe and doomed regions we simply apply the algorithm on the unfolding.

We start by introducing formally the core-cutting operators before extending the lifting property of Theorem 4.3.28 to the generic cubical partition, proving that the generic cubical partition of the base program corresponds to the quotient of the cubical partition of the unfolding.

Finally, we prove that there is an equivalence for the relation "being in the past of" (Definition 1.4.31) between cubes of the base program and their lifting.

All this allows us to implement a modified version of Algorithm 3.3.23 on the unfolding of a program, which allows us to detect deadlocks in the base programs (which potentially has loops).

4.4.1 Cubical partition and loops

In this section we will prove that the lifting property of Theorem 4.3.27 extends to the cubes of the generic partition (Definition 4.1.1) of the maximal cover. We recall that this generic partition is the one we use to represent the set of positions in Algorithm 3.3.23 and compute deadlocks.

Definition 4.4.3. Given a cover R on a sequential program P, we define the 1-*dimensional generic cubical partition* as

$$\Gamma_1^m(R) = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} \mathcal{C}^{\max}(Z_{U,V})$$

with $Z_{U,V} = \bigcap_{u \in U} [u] \bigcap_{v \in V} [v]^{\mathrm{c}}$

Definition 4.4.4. Given a cover R on the parallel composition of n sequential programs $P = P_1 \| \cdots \| P_n$, we define the *generic cubical partition* as

$$\Gamma^*(R) = \Gamma^m \circ \Gamma^m(R)$$

where $\Gamma^m(R)$ is defined as

$$\Gamma^m(R) = \prod_{i=1}^{n} \Gamma_1^m\big(\{c_i \mid \exists (r_k)_{1 \leq k \leq n} \in R, r_i = c_i\}\big)$$

Remark 4.4.5. For now, we do not prove that the generic partition is a partition. That is not necessary as it will be a consequence of Theorem 4.4.45.

As seen in Example 4.4.1, we cannot directly prove that the lifting and the generic partition commute. In order to enforce the commutation, we define what we call the core-cutting operators, that separates the cubes of the partition of the unfolding in order to replicate what happens at the core of the loop in the base program.

Remark 4.4.6. Given a program P^*, there is one maximal conservative cubical cover that does not get cut when taking its cubical partition: that is the cover $R = \{(\bot, \top)\}$. Here, we will cut this cube in the unfolding, meaning that we will need to cut it in the final partition. This should not be needed, but no proof circumventing this problem could be found.

The idea is that for a loop, as shown in Example 4.4.1, conservative regions crossing the core of the loop $p^\star$ are separated in four disjoint regions by the generic partition: before the loop, at the core of the loop, strictly inside the loop and after the loop.

In the unfolding, as the partition does not provide this kind of action, we need to do it ourselves. Where we need to cut is given by the Example 4.4.1 and is quite naturally

at the lifting of the core $\top^*$, thus separating the cubes of the lifting in exactly the right way.

This cutting operation is formalized through the operator Θ for the unfolding and χ for the base program.

Definition 4.4.7. Given a program $P \in \mathbf{Prcs}^*$, we define the *unfolded-core-cutting* operator Θ_P on cubes of P as follows.

$$\begin{aligned}\Theta_{\mathcal{U}(P)} \colon \mathcal{C}(\mathcal{U}(P)) &\to \mathcal{R}(\mathcal{U}(P)) \\ c &\mapsto \begin{cases} \mathcal{C}^{\max}\big([c] \setminus \Phi^{-1}(\top^*)\big) & \text{if } P = Q^* \\ \bigcup\limits_{x \in \Phi^{-1}(\top^*)} \mathcal{C}^{\max}\big([c] \cap \{x\}\big) & \\ \{c\} & \text{otherwise} \end{cases}\end{aligned}$$

And for parallel composition $P = \mathcal{U}(P_1) \| \mathcal{U}(P_2)$ we define

$$\begin{aligned}\Theta_{\mathcal{U}(P_1)\|\mathcal{U}(P_2)} \colon \mathcal{C}(\mathcal{U}(P_1)\|\mathcal{U}(P_2)) &\to \mathcal{R}(\mathcal{U}(P_1)\|\mathcal{U}(P_2)) \\ c_1 \| c_2 &\mapsto \Theta_{\mathcal{U}(P_1)}(c_1) \times \Theta_{\mathcal{U}(P_2)}(c_2)\end{aligned}$$

Then we can easily extend Ω_P to regions of P as follows.

$$\begin{aligned}\Theta_{\mathcal{U}(P)} \colon \mathcal{R}(\mathcal{U}(P)) &\to \mathcal{R}(\mathcal{U}(P)) \\ \{c_i \mid i \in I\} &\mapsto \bigcup_{i \in I} \Theta_{\mathcal{U}(P)}(c_i)\end{aligned}$$

Definition 4.4.8. Given a program $P \in \mathbf{Prcs}^*$, we define the $\Pi_P \mathcal{L}_{-1}$ *core-cutting* operator χ_P on cubes of P as follows.

$$\begin{aligned}\chi_P \colon \mathcal{C}(P) &\to \mathcal{R}(P) \\ c &\mapsto \begin{cases} \mathcal{C}^{\max}\big([c] \setminus \{\top^*\}\big) & \text{if } c = (\bot, \top) \text{ and } P = Q^* \\ \quad \cup\, \mathcal{C}^{\max}\big([c] \cap \{\top^*\}\big) & \\ \{c\} & \text{otherwise} \end{cases}\end{aligned}$$

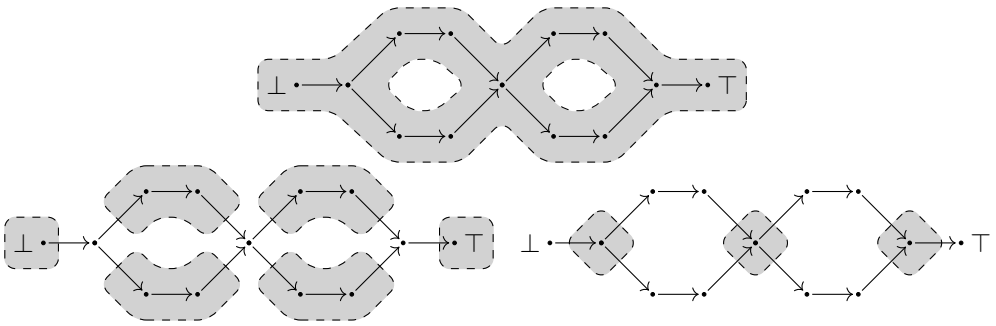
And for parallel composition $P = P_1 \| P_2$ we define

$$\begin{aligned}\chi_{P_1\|P_2} \colon \mathcal{C}(P_1\|P_2) &\to \mathcal{R}(P_1\|P_2) \\ c_1 \| c_2 &\mapsto \chi_{P_1}(c_1) \times \chi_{P_2}(c_2)\end{aligned}$$

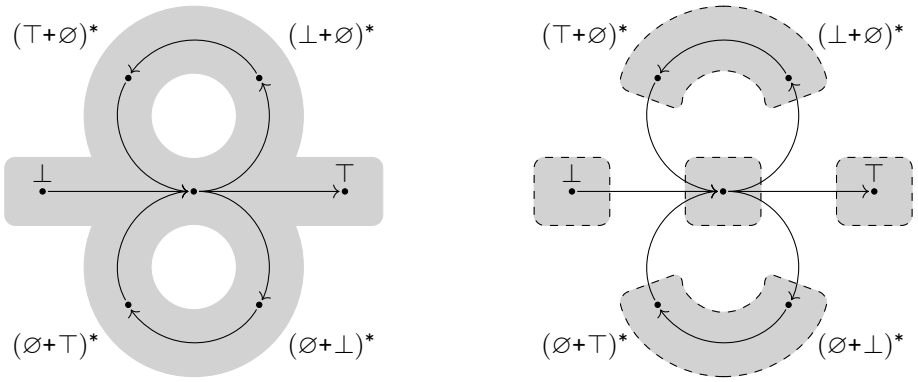
Then we can easily extend χ_P to regions of P as follows.

$$\begin{aligned}\chi_P \colon \mathcal{R}(P) &\to \mathcal{R}(P) \\ \{c_i \mid i \in I\} &\mapsto \bigcup_{i \in I} \chi_P(c_i)\end{aligned}$$

Example 4.4.9. Let us consider the program `(skip+skip);(skip+skip)` whose semantics are given below. On the left we see the result of $\mathcal{C}^{\max}\big([\bot, \top] \setminus \Phi^{-1}(\top^*)\big)$ and on the right the result of $\bigcup\limits_{x \in \Phi^{-1}(\top^*)} \mathcal{C}^{\max}\big([\bot, \top] \cap \{x\}\big)$



Example 4.4.10. Let us consider the cube $(\bot, \top)$ on the program $(\texttt{skip+skip})^*$ which unfolds to the program of Example 4.4.9. Its semantics are shown on the left with the support of $(\bot, \top)$ in grey. The result by the operator χ is given on the right.



If Θ left the cube $(\bot, \top)$ untouched, the operator χ would be totally unnecessary, as indeed the generic partition of the lifting is equal to $(\bot, \top)$ which is equal to the lifting of the generic partition. Unfortunately, this breaks the fact that cubes included in the support of a cube in the image of Θ are stable by Θ, which we needed in our proof of Theorem 4.4.45 (more precisely Proposition 4.4.37 and accompanying lemmas).

With this choice, we can now formulate the first main theorem of this chapter

Theorem 4.4.45. *Given a program $P \in$ **Pgrm** and a maximal conservative cover R on P we have*

$$\Theta \circ \Gamma^* \circ \Psi^{-1}(R) = \Psi^{-1} \circ \Gamma^*(\chi(R))$$

This is a tall order, so we break down the proof in smaller proofs, each time proving a step of the commutation.

We also introduce a third separation operator Ω that performs the same separation as χ but to all cubes.

Definition 4.4.11. Given a program $P \in$ **Prcs**, we define the *core-cutting* operator Ω_P on cubes of P as follows.

$$\Omega_{P^\star} \colon \mathcal{C}(P^\star) \to \mathcal{R}(P^\star)$$
$$(p,q) \mapsto \begin{cases} \mathcal{C}^{\max}\big([c] \setminus \{\top^\star\}\big) & \text{if } c = (p,q) \\ \quad \cup\, \mathcal{C}^{\max}\big([c] \cap \{\top^\star\}\big) & \\ \Omega(p,\top^\star) \cup \Omega(\top^\star, q) & \text{if } c = (p,\top^\star,q) \end{cases}$$

Otherwise, when not dealing with loops, Ω sends cube to the region containing only that cube.

$$\begin{aligned}\Omega_P \colon \mathcal{C}(P) &\to \mathcal{R}(P)\\ c &\mapsto \{c\}\end{aligned}$$

And for parallel composition $P = P_1 \| P_2$ we define

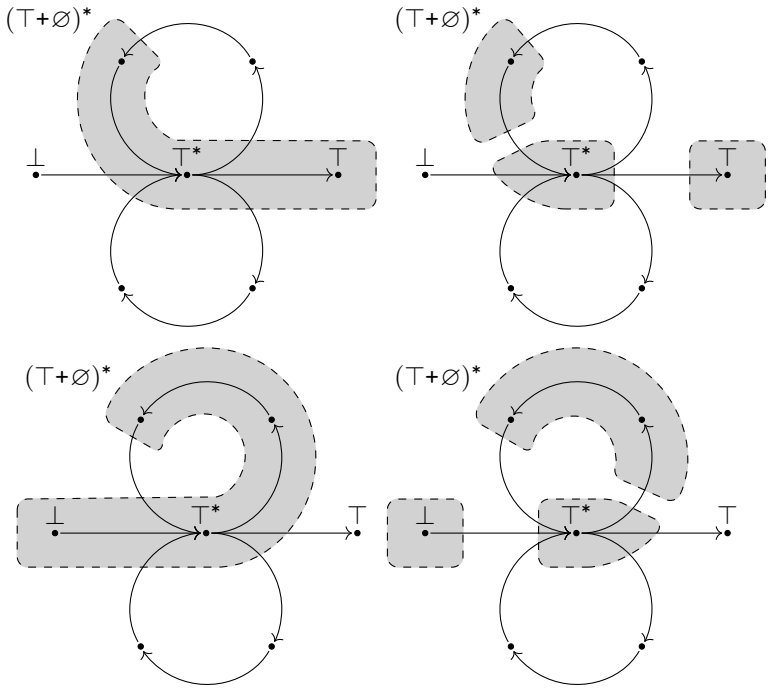
$$\begin{aligned}\Omega_{P_1\|P_2} \colon \mathcal{C}(P_1\|P_2) &\to \mathcal{R}(P_1\|P_2)\\ c_1\|c_2 &\mapsto \Omega_{P_1}(c_1) \times \Omega_{P_2}(c_2)\end{aligned}$$

Then we can easily extend Ω_P to regions of P as follows.

$$\begin{aligned}\Omega_P \colon \mathcal{R}(P) &\to \mathcal{R}(P)\\ \{c_i \mid i \in I\} &\mapsto \bigcup_{i\in I} \Omega_P(c_i)\end{aligned}$$

Remark 4.4.12. For all operators we omit the subscript when it is obvious from the context.

Example 4.4.13. Let us consider the program $(\texttt{skip+skip})^*$ whose semantics is given below. We show the action of the operator Ω on the cubes $(\bot, (\top+\varnothing)^*)$ and $((\top+\varnothing)^*, \bot)$.

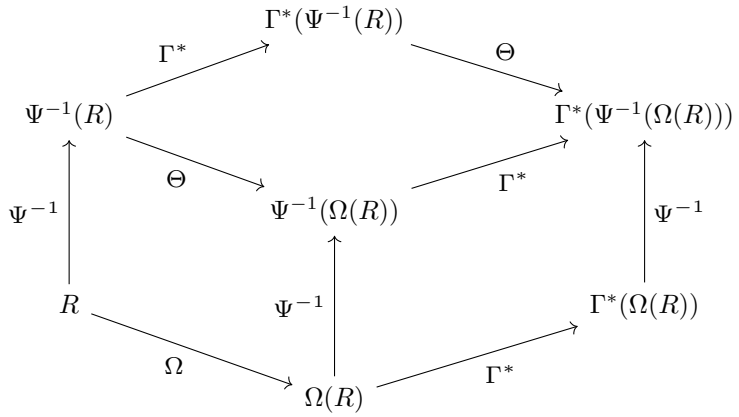


This operator will be very useful in the proof of Theorem 4.4.45. Indeed instead of proving Theorem 4.4.45 we can first prove the weaker Proposition 4.4.38.

Proposition 4.4.38. *Given a program $P \in \mathbf{Pgrm}$ and a maximal conservative cover R on P we have*

$$\Theta \circ \Gamma^* \circ \Psi^{-1}(R) = \Psi^{-1} \circ \Gamma^* \Omega(R)$$

The proof of Proposition 4.4.38 will consists in proving the commutativity of the following diagram, in which we will prove all commutative squares individually.



We define an object that appears in most of our proofs. This corresponds to the cover of one of the elements of our cubical partition. This was already introduced in previous section, but is reminded here as it will be frequently used in the following proofs.

Definition 4.4.14. For a partition $U \amalg V = R$ of a cover R, the associated *support separation* $Z_{U,V}$ is defined as follows

$$Z_{U,V} = \bigcup_{u \in U} [u] \bigcup_{v \in V} [v]^c$$

4.4.2 Cubical partition as a quotient of the 2-unfolding

4.4.2.1 $\Psi^{-1} \circ \Omega = \Theta \circ \Psi^{-1}$

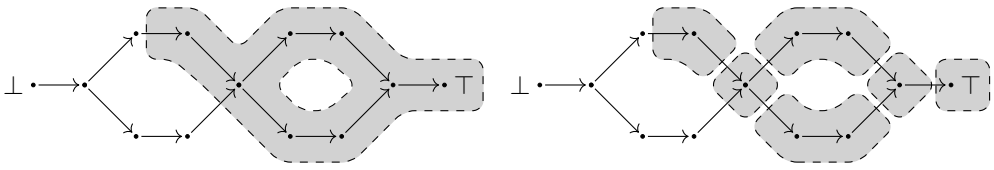
In this section we will prove the commutation of the lifting with the separation operators

Proposition 4.4.21. *Given a program $P \in \mathbf{Prcs}^*$, R a maximal conservative cover on P. Then*

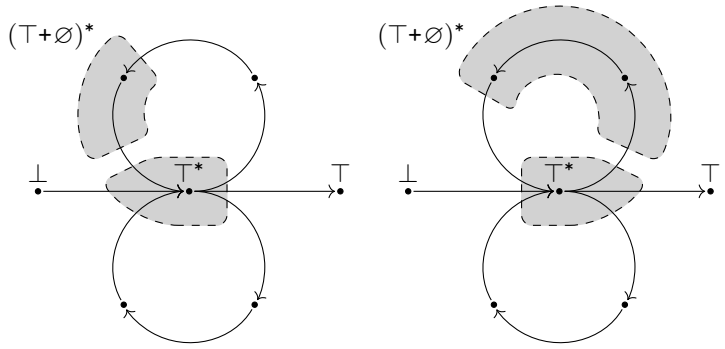
$$\Psi^{-1} \circ \Omega(R) = \Theta \circ \Psi^{-1}(R)$$

In order to do so, we will fully characterize the image of Θ and Ω in Proposition 4.4.17 and Proposition 4.4.18. Indeed, Θ actually creates a cubical partition when cutting the cubes, and those can be computed by looking at the successors and predecessors of the core's lifting, which there is only one pair of that creates a non-empty-cube w.r.t. the cube being cut.

Example 4.4.15. Let us look at the action of theta on some of the more problematic cubes of the program $P = \mathcal{U}((\texttt{skip+skip})^*)$. The cube $((\top + \varnothing); \bot, \top)$ shown below on the left is properly partitioned by theta as shown on the right. The fact that a copy of the loop is separated in disjoint is a consequence of Lemma 4.4.16 as explained above.



Unfortunately, such a property doesn't hold for Ω as the separation of composite cubes might lead to some overlapping as shown below on the cube $((\top + \varnothing)^*, \top^*, (\top + \varnothing)^*)$



Then Proposition 4.4.20 allows us to link the cubes of Θ and Ω through these unique antecedents.

Lemma 4.4.16. *Let $P \in \mathbf{Prcs} \setminus \alpha$. Let $p, q \in \mathcal{P}(P) \setminus \{\bot, \top\}$. Then*

- *There exists a unique element $\overleftarrow{\top}_p \in \mathsf{pred}\top$, such that a path $p \to^* \overleftarrow{\top}_p \to \top$ exists and is without trivial loops.*
- *There exists a unique element $\vec{\bot}_p \in \mathsf{succ}\bot$, such that a path $\bot \to \vec{\bot}_q \to q$ exists and is without trivial loops.*

Proof. Let $P \in \mathbf{Prcs} \setminus \alpha$. Let $p, q \in \mathcal{P}(P) \setminus \{\bot, \top\}$. Let us prove the property for $\overleftarrow{\top}_p$, the other property being dual.

- If $P = S;T$. Then $\mathsf{pred}\top = \top;\top$. And it is easy to show that $p < \top$ implies $p \leq \top;\top \leq \top$. By Proposition 4.1.32, this implies $p \to^* \top;\top \to \top$. As the program P is loop-free, the path is loop-free and $\overleftarrow{\top}_p = \top;\top$.
- If $P = S+T$. Then $\mathsf{pred}\top = \{\varnothing+\top, \top+\varnothing\}$. $\bot < p < \top$ implies $p = s+\varnothing$ or $p = \varnothing+t$.
  - If $p = s+\varnothing$ Then $s+\varnothing \not\leq \varnothing+\top$. Thus, by Proposition 4.1.32, there is no path $s+\varnothing \to^* \varnothing+\top$. By the same argument, as $s+\varnothing \leq \top+\varnothing$ there is a path $s+\varnothing \to^* \top+\varnothing$. As the program P is loop-free so are all the paths. Thus, $\overleftarrow{\top}_p = \top+\varnothing$.
  - If $p = \varnothing+t$, then, by a similar argument $\overleftarrow{\top}_p = \varnothing+\top$.

□

Proposition 4.4.17. *Given a program $\mathcal{U}(P^\star)$, and two positions p, q of $\mathcal{U}(P^\star)$, such that $p, q \notin \{\bot, \top\}$ and a cube $c \in \mathcal{C}_0$ we have*

$$\Theta(p;\bot,q;\bot) = \{(p;\bot,q;\bot)\} \qquad \Theta(\top;p,\top;q) = \{(\top;p,\top;q)\}$$
$$\Theta(\bot,\bot;\bot) = \{(\bot,\bot),(\bot;\bot,\bot;\bot)\} \qquad \Theta(\top;\top,\top) = \{(\top,\top),(\top;\top,\top;\top)\}$$
$$\Theta(\bot;\bot,q;\bot) = \{(\bot;\bot,\bot;\bot),(\vec{\bot}_q;\bot,q;\bot)\} \qquad \Theta(\top;\bot,\top;q) = \{(\top;\bot,\top;\bot),(\top;\vec{\bot}_q,\top;q)\}$$
$$\Theta(p;\bot,\top;\bot) = \{(\top;\bot,\top;\bot),(p;\bot,\overleftarrow{\top}_p;\bot)\} \qquad \Theta(\top;p,\top;\top) = \{(\top;\top,\top;\top),(\top;p,\top;\overleftarrow{\top}_p)\}$$
$$\Theta(c) = \{c\}$$

$$\Theta(\bot,\top) = \mathcal{C}_0 \cup \{(x;\bot,\overleftarrow{\top}_x;\bot),(\top;x,\top;\overleftarrow{\top}_x) \mid x \in \mathsf{succ}\bot\}$$

Proof. Given a program $\mathcal{U}(P^\star)$, we have the first three cases directly by definition of Θ.

- The cases $c \in \mathcal{C}_0$, $(\bot,\bot;\bot)$, $(\top;\top,\top)$ are easily deduced from the definition of Θ.
- Let us consider the case $(p;\bot,q;\bot)$. Then, we have $[p;\bot,q;\bot] \subseteq \{\top^\star\}^c$. Thus, by definition of Θ, $\Theta(p;\bot,q;\bot) = \mathcal{C}^{\max}([p;\bot,q;\bot]) = \{(p;\bot,q;\bot)\}$
- The case $(\top;p,\top;q)$ is dual to the previous case.
- For the case $(\bot,\top)$, we first prove

  $$\Theta(\bot;\bot,\top;\bot)\{(\bot;\bot,\bot;\bot),(\top;\bot,\top;\bot)\} \cup \{(x;\bot,\overleftarrow{\top}_x;\bot) \mid x \in \mathsf{succ}\bot\}$$

  Indeed, by Lemma 4.4.16 for any $p;\bot,q;\bot$ there exist a unique pair, x, y such that $\bot \to x \to^* p \to^* q \to^* y \to \top$. Furthermore, $y = \overleftarrow{\top}_x;\bot$ and by inference rules we can show that the cubes $(x;\bot,\overleftarrow{\top}_x;\bot)$ are exactly the maximal cubes of $[\bot;\bot,\top;\bot] \setminus \{\bot;\bot,\top;\bot\}$. Thus,

  $$\Theta(\bot;\bot,\top;\bot)\{(\bot;\bot,\bot;\bot),(\top;\bot,\top;\bot)\} \cup \{(x;\bot,\overleftarrow{\top}_x;\bot) \mid x \in \mathsf{succ}\bot\}$$

  Similarly

  $$\Theta(\top;\bot,\top;\top)\{(\bot;\bot,\bot;\bot),(\top;\bot,\top;\bot)\} \cup \{(\top;x,\top;\overleftarrow{\top}_x) \mid x \in \mathsf{succ}\bot\}$$

  then we conclude by remarking that we can decompose $(\bot,\top)$ as the composite of the cubes $(\bot,\bot;\bot)$, $(\bot;\bot,\top;\bot)$, $(\top;\bot,\top;\top)$ and $(\top;\top,\top)$.
- $(\bot;\bot,q;\bot)$ follows a similar proof to the case $\Theta(\bot;\bot,\top;\bot)$ in the case above, where we take $y = q$.
- The case $(\top;\bot,\top;q),(p;\bot,\top;\bot),(\top;p,\top;\top)$ are all treated similarly to the case above.

□

Proposition 4.4.18. *Given a program $\mathcal{U}(P^\star)$, two positions p, q of $\mathcal{U}(P^\star)$, such that $p, q \notin \{\bot, \top\}$, and a cube $c \in \mathcal{L}_0$, we have*

$$\Omega(c) = \{c\} \qquad \Omega(p^\star, q^\star) = \{(p^\star, q^\star)\}$$
$$\Omega(\bot, \top^\star) = \{(\top^\star, \top^\star),(\bot,\bot)\} \qquad \Omega(p^\star, \top^\star) = \{(p^\star, \overleftarrow{\top}_p{}^*),(\top^\star,\top^\star)\}$$
$$\Omega(\top^\star, \top) = \{(\top^\star,\top^\star),(\top,\top)\} \qquad \Omega(\top^\star, q^\star) = \{(\vec{\bot}^*_q, q^\star),(\top^\star,\top^\star)\}$$

$$\Omega(\bot,\top) = \{(\bot,\bot),(\top^\star,\top^\star),(\top,\top)\} \cup \{(x^\star, \overleftarrow{\top}_x{}^*) \mid x \in \mathsf{succ}\top\}$$

Proof. Given a program $\mathcal{U}(P^\star)$, we have the first three cases directly by definition of Ω.

- The cases $c \in \mathcal{L}_0$, $(\bot, \top^\star)$, $(\top^\star, \top)$ are easily deduced from the definition of Ω.
- Let us consider the case $(p^\star, q^\star)$. Then, we have $[p^\star, q^\star] \subseteq \{\top^\star\}^c$. Thus, by definition of Ω, $\Omega(p^\star, q^\star) = \mathcal{C}^{\max}([p^\star, q^\star]) = \{(p^\star, q^\star)\}$
- Let us consider the case $(p^\star, \top^\star)$. For all paths $\pi \in (p^\star, \top^\star)$, as $p \neq \bot, \top$, we have $\pi = \sigma^* \cdot \tau^*$ such that
$$\sigma^* \colon p^\star \to^+ x^\star \text{ and } \tau^* \colon x^\star \to \top^\star$$
By Definition 4.2.13, we have $\sigma \colon p \to^+ x$ and by inference rules $\tau \colon x \to \top$. Thus, there is a loop-free path $p \to^* x \to \top$. By Lemma 4.4.16 $x = \overleftarrow{\top}_p$. This implies
$$[p^\star, \top^\star] = [p^\star, \overleftarrow{\top}_p{}^*] \cup \{\top^\star\}$$
Thus by definition of Ω,
$$\Omega(p^\star, \top^\star) = \mathcal{C}^{\max}([p^\star, \overleftarrow{\top}_p{}^*]) \cup \mathcal{C}^{\max}(\{\top^\star\})$$
$$\Omega(p^\star, \top^\star) = \{(p^\star, \overleftarrow{\top}_p{}^*)\} \cup \{(\top^\star, \top^\star)\}$$
- $(\top^\star, q^\star)$. Dual to the case above.
- Let us consider the case $(\bot, \top)$. If $P = \alpha$ the proposition holds as $\{(x^\star, \overleftarrow{\top}_x{}^*) \mid x \in \mathsf{succ}\top\}$ is empty. Thus, for the rest of the proof, we will suppose
$$\mathcal{P}(P) \setminus \{\bot, \top\} \neq \emptyset$$
then, by definition, for any pair of positions $(p, q) \in \mathcal{P}(P) \setminus \{\bot, \top\}$, there is a pair $x_p, y_q \in \mathsf{succ}\bot \times \mathsf{pred}\top$ such that $(p, q) \subseteq (x_p, y_q)$. Thus, by Lemma 4.3.16,
$$(p^\star, q^\star) \subseteq (x_p^\star, y_q^\star)$$
Furthermore, by Lemma 4.4.16, $y_q = \overleftarrow{\top}_{x_p}$, and by inference rules $x_p \in \mathsf{succ}\top$ This implies by Lemma 4.2.14 that $\mathcal{C}^{\max}(\mathcal{P}(P^\star) \setminus \{\bot, \top, \top^\star\}) \subseteq \{(x^\star, \overleftarrow{\top}_x{}^*) \mid x \in \mathsf{succ}\top\}$ which concludes the proof.

□

Remark 4.4.19. The Proposition 4.4.18 and Proposition 4.4.17 also imply that Θ and Ω can be computed if we know how to find the successors and predecessors of points as well as intersection of cubes. This is much quicker in practice than using the definition of θ which requires the calculation of many complements and intersection to obtain the same result.

Proposition 4.4.20. *Let $P \in \mathbf{Prcs} \setminus \alpha$. Let $p, q \in \mathcal{P}(P) \setminus \{\bot, \top\}$*
$$\Psi^{-1}(p^\star, \overleftarrow{\top}_p{}^*) = \{(\top\,;p, \top\,;\overleftarrow{\top}_p), (p\,;\bot, \overleftarrow{\top}_p\,;\bot)\}$$
$$\Psi^{-1}(\vec{\bot}_q^*, q^\star) = \{(\top\,;\vec{\bot}_q, \top\,;q), (\vec{\bot}_q\,;\bot, q\,;\bot)\}$$

Proof. By definition $\overleftarrow{\top}_p, \overrightarrow{\bot}_q \notin \{\bot, \top\}$. Thus, $(p^\star, \overleftarrow{\top}_p{}^*)$, $(\overrightarrow{\bot}_q^*, q^\star) \in \mathcal{L}_5$. By Proposition 4.2.36, we obtain our result. □

Proposition 4.4.21. *Given a program $P \in \mathbf{Prcs}^*$, R a maximal conservative cover on P. Then*

$$\Psi^{-1} \circ \Omega(R) = \Theta \circ \Psi^{-1}(R)$$

Proof. By induction on P

- $P \in \mathbf{Prcs}$. OK.
- $P = P_1 \parallel P_2$. By Proposition 4.3.7, for all $c = c_1 \times c_2 \in R$ the sets

  $$\pi_1 R_c = \{c' \in \mathcal{P}(P_1) \mid c' \times c_2 \in R\} \text{ and } \pi_1 R_c = \{c' \in \mathcal{P}(P_2) \mid c_1 \times c' \in R\}$$

  are conservative and maximal and $R = \bigcup_{c \in R} \pi_1 R_c \times \pi_2 R_c$ Thus,

  $$\Psi^{-1} \circ \Omega(R) = \bigcup_{c \in R} \Psi^{-1} \circ \Omega(\pi_1 R_c) \times \Psi^{-1} \circ \Omega(\pi_2 R_c)$$

  $\bigcup_{c \in R} \Theta \circ \Psi^{-1}(\pi_1 R_c) \times \Theta \circ \Psi^{-1}(\pi_2 R_c)$by induction hypothesis

  $$\Psi^{-1} \circ \Omega(R) = \Theta \circ \Psi^{-1}(R)$$

- $P = Q^\star$. Then by definition of R,
  - $\top^\star \notin [R]$. Then, by Proposition 4.3.21,

    $$R = \{(p_i^\star, q_i^\star) \mid \forall i \in I, p_i, q_i \neq \bot, \top\}$$

    As $\top^\star \notin [R]$, a fortiori for all $i \in I$, $\top^\star \notin [(p_i^\star, q_i^\star)]$, i.e. $\Omega((p_i^\star, q_i^\star)) = \{(p_i^\star, q_i^\star)\}$. Thus, for all $i \in I$

    $$\begin{aligned} \Psi^{-1}(\Omega(p_i^\star, q_i^\star)) &= \Psi^{-1}(p_i^\star, q_i^\star) \\ &= \{(p_i ; \bot, q_i ; \bot), (\top ; p_i, \top ; q_i)\} \\ &= \Theta(\{(p_i ; \bot, q_i ; \bot), (\top ; p_i, \top ; q_i)\}) \qquad \text{as } p_i, q_i \neq \bot, \top \\ \Psi^{-1}(\Omega(p_i^\star, q_i^\star)) &= \Theta(\Psi^{-1}(p_i^\star, q_i^\star)) \end{aligned}$$

    Thus, $\Psi^{-1} \circ \Omega(R) = \Theta \circ \Psi^{-1}(R)$
  - $R \neq \{(\bot, \top)\}$ and $\top^\star \in [R]$. Then by Proposition 4.3.21,

    $$R = \bigcup_{j \in J_1} \{(p_j^\star, q_j^\star)\} \bigcup_{j \in J_2} \{(\bot, q_j^\star), (p_j^\star, \top), (p_j^\star, \top^\star, q_j^\star), (\bot, \top^\star, \top)\}$$

    Such that $p_j^\star, q_j^\star \neq \top^\star$. By the same argument as the case above we have,

    $$\Psi^{-1} \circ \Omega(\bigcup_{j \in J_1} \{(p_j^\star, q_j^\star)\}) = \Theta \circ \Psi^{-1}(\bigcup_{j \in J_1} \{(p_j^\star, q_j^\star)\})$$

Then, for each $j \in J_2$, we write

$$R_j = \{(\bot, q_j^*), (p_j^*, \top), (p_j^*, \top^*, q_j^*), (\bot, \top^*, \top)\}$$

Then

* $\Omega(\bot, q_j^*) = \{(\bot, \bot), (\top^*, \top^*), (\vec{\bot}^*_{q_j}, q_j^*)\}$
* $\Omega(p_j^*, \top) = \{(\top, \top), (\top^*, \top^*), (p_j^*, \overleftarrow{\top}_{p_j}{}^*)\}$
* $\Omega(p_j^*, \top^*, q_j^*) = \{(\top^*, \top^*), (p_j^*, \overleftarrow{\top}_{p_j}{}^*), (\vec{\bot}^*_{q_j}, q_j^*)\}$
* $\Omega(\bot, \top^*, \top) \subseteq \Omega(R_j \setminus (\bot, \top^*, \top))$

With $\Psi^{-1}(R_j) = \{(p; \bot, \top; q), (\bot, q; \bot), (\top; p, \top)\}$, we have

* $\Theta(\bot, q; \bot) = \{(\bot, \bot), (\bot; \bot, \bot; \bot), (\vec{\bot}_q; \bot, q; \bot)\}$
* $\Theta(\top; p, \top) = \{(\top, \top), (\top; \top, \top; \top), (\top; p, \top; \overleftarrow{\top}_p)\}$
* $\Theta(p; \bot, \top; q) = \{(p; \bot, \overleftarrow{\top}_p; \bot), (\top; \bot, \top; \bot), (\top; \vec{\bot}_q, \top; q)\}$

Now as we have that

* By Proposition 4.4.20, $\Psi^{-1}(p^*, \overleftarrow{\top}_p{}^*) = \{(p; \bot, \overleftarrow{\top}_p; \bot), (\top; p, \top; \overleftarrow{\top}_p)\}$
* By Proposition 4.4.20, $\Psi^{-1}(\vec{\bot}^*_q, q^*) = \{(\vec{\bot}_q; \bot, q; \bot), (\top; \vec{\bot}_q, \top; q)\}$
* $\Psi^{-1}(\top^*, \top^*) = \{(\bot; \bot, \bot; \bot), (\top; \bot, \top; \bot), (\top; \top, \top; \top)\}$
* $\Psi^{-1}(\bot, \bot) = (\bot, \bot)$ and $\Psi^{-1}(\top, \top) = (\top, \top)$

We can conclude

$$\Psi^{-1} \circ \Omega(R_j) = \Theta \circ \Psi^{-1}(R_j)$$

$$\begin{aligned}
\Psi^{-1} \circ \Omega(R) &= \bigcup_{j \in J_1} \Psi^{-1} \circ \Omega(R)\{(p_j^*, q_j^*)\} \bigcup_{j \in J_2} \Psi^{-1} \circ \Omega(R)(R_j) \\
&= \bigcup_{j \in J_1} \Theta \circ \Psi^{-1}\{(p_j^*, q_j^*)\} \bigcup_{j \in J_2} \Theta \circ \Psi^{-1}(R_j) \\
\Psi^{-1} \circ \Omega(R) &= \Theta \circ \Psi^{-1}(R)
\end{aligned}$$

– $R = \{(\bot, \top)\}$. By Proposition 4.4.18, we have

$$\Omega(\bot, \top) = \{(\bot, \bot), (\top, \top), (\top^*, \top^*)\} \cup \{(\vec{\bot}^*_q, q^*) \mid q \to \top\}$$

And Proposition 4.4.17, we have

$$\begin{aligned}
\Theta(\bot, \top) = \{&(\bot, \bot), (\top, \top), (\bot; \bot, \bot; \bot), (\top; \bot, \top; \bot), (\top; \top, \top; \top)\} \\
&\cup \{(\vec{\bot}_q; \bot, q; \bot), (\top; \vec{\bot}_q, \top; q) \mid q \to \top\}
\end{aligned}$$

Using the same argument as the case above, we have thus,

$$\Psi^{-1} \circ \Omega(R) = \Theta \circ \Psi^{-1}(R)$$

□

4.4.2.2 $\Gamma^* \circ \Theta = \Theta \circ \Gamma^*$

In this section we prove the commutation of the separation operator Θ and the generic partition.

Proposition 4.4.29. *Let $P \in \mathbf{Pgrm}$, for any conservative cover R on a program $\mathcal{U}(P)$,*

$$\Theta \circ \Gamma^*(R) = \Gamma^* \circ \Theta(R)$$

We begin by stating a few of the important properties about theta, namely, the fact that Θ defines a cubical partition of the support of cubes it cuts and that it preserves and reflects inclusion of cubes to a degree.

Lemma 4.4.22. *Given a program $P \in \mathbf{Pgrm}$ and a cube $c \in \mathcal{C}(\mathcal{U}(P))$, we have*

$$[\Theta(c)] = \coprod_{d \in \Theta_P(c)} [d] = [c]$$

Proof. By induction on P.

- $P \in \mathbf{Prcs}$. Then Θ is the identity.
- $P = P_1 \parallel P_2$. Then, $c \in \mathcal{C}(\mathcal{U}(P))$ implies $c = c_1 \times c_2$ and

$$\begin{aligned} [\Theta(c)] &= [\Theta(c_1)] \times [\Theta(c_2)] \\ &= \coprod_{d_1 \in \Theta_P(c_1)} [d_1] \times \coprod_{d_2 \in \Theta_P(c_2)} [d_2] && \text{induction hypothesis} \\ [\Theta(c)] &= \coprod_{d \in \Theta_P(c)} [d] \end{aligned}$$

  Furthermore,

$$\begin{aligned} [\Theta(c)] &= [\Theta(c_1)] \times [\Theta(c_2)] \\ &= [c_1] \times [c_2] && \text{induction hypothesis} \\ [\Theta(c)] &= [c] \end{aligned}$$

  Which concludes the proof.

- $P = Q^\star$. Then by definition of Θ,

$$\begin{aligned} [\Theta(c)] &= [\mathcal{C}^{\max}\big([c] \setminus \Phi^{-1}(\top^\star)\big) \bigcup_{x \in \Phi^{-1}(\top^\star)} \mathcal{C}^{\max}\big([c] \cap \{x\}\big)] \\ [\Theta(c)] &= [c] \setminus \Phi^{-1}(\top^\star) \bigcup_{x \in \Phi^{-1}(\top^\star)} [c] \cap \{x\}[\Theta(c)] && = [c] \end{aligned}$$

  The fact that all cubes of $\Theta(c)$ are disjoint is a direct consequence of Proposition 4.4.17. The only case not directly treated $c \in \mathcal{C}_{-1}$ can be done so in the same manner as the case $(\bot, \top)$.

□

Lemma 4.4.23. *Given a program $P \in \mathbf{Pgrm}$ and a cube $u \in \Theta(\mathcal{C}(P))$, then we have*

- $\Theta_P \circ \Theta_P = \Theta_P$
- *For any subset $X \subseteq [u]$, $\mathcal{C}(X) \in \Theta(\mathcal{C}(P))$*

Proof. This holds directly by definition of Θ. □

Lemma 4.4.24. *Given a program $P \in \mathbf{Pgrm}$ and $c, d \in \mathcal{C}(\mathcal{U}(P))$, then $c \subseteq d$ implies for all $x \in \Theta(c)$, there exists $y \in \Theta(d)$ such that $x \subseteq y$*

Proof. Let us prove the property by induction on P

- $P \in \mathbf{Prcs}$. Then Θ is the identity.
- $P = P_1 \,\|\, P_2$. Let $c = c_1 \times c_2 \subseteq d_1 \times d_2 = d$. Then $c_1 \subseteq d_1$ and $c_2 \subseteq d_2$. By induction hypothesis, this is implies for all $x_1, x_2 \in \Theta(c_1) \times \Theta(c_2) = \Theta(c)$, there exists $y_1 \times y_2 \in \Theta(d_1) \times \Theta(d_2) = \Theta(d)$ such that $x_1 \times x_2 \subseteq y_1 \times y_2$. Which concludes the proof.
- $P = Q^*$. Then by definition of Θ,

$$\begin{aligned}\Theta(c) &= \mathcal{C}^{\max}\big([c] \setminus \{\Phi^{-1}(\top^*)\}\big) \bigcup_{x \in \Phi^{-1}(\top^*)} \mathcal{C}^{\max}\big([c] \cap \{x\}\big) \\ &\leq \mathcal{C}^{\max}\big([d] \setminus \{\Phi^{-1}(\top^*)\}\big) \bigcup_{x \in \Phi^{-1}(\top^*)} \mathcal{C}^{\max}\big([d] \cap \{x\}\big) \\ \Theta(c) &\leq \Theta(q)\end{aligned}$$

  where $\leq$ is the order on region defined in Definition 4.2.46. Thus, the property holds. □

Lemma 4.4.25. *Given a conservative cover X on a program $\mathcal{U}(P)$, $P \in \mathbf{Prcs}^*$, such that $\mathcal{C}(X) \cap \mathcal{C}_{-1} = \emptyset$. Given $c \in \mathcal{C}(X)$, $u \in \Theta(c)$. Then $u \subseteq v \in \Theta(\mathcal{C}(X))$ implies that there exists $d \in \mathcal{C}(X)$, such that $c \subseteq d$*

Proof. By induction on P.

- $P \in \mathbf{Prcs}$. Then, θ is the identity on all cubes, concluding the proof.
- $P = L^*$, such that $\mathcal{U}(P) = L;L$. Given $s \in \Theta(\mathcal{C}(X))$,

$$s \in \mathcal{C}_0 \cup \mathcal{C}(\{q;\bot \mid q \in \mathcal{P}(L) \setminus \{\bot, \top\}\}) \cup \mathcal{C}(\{\top;q \mid q \in \mathcal{P}(L) \setminus \{\bot, \top\}\})$$

  Then as $u \in \Theta(c) \subseteq \Theta(\mathcal{C}(X))$ and $v \in \Theta(\mathcal{C}(X))$ we can separate the different cases:

  - $u \in \mathcal{C}_0$. Then $v \neq u$ implies $u \not\subseteq v$. Thus, $v = u$ and $d = c$ verifies the conditions.
  - $u \in \mathcal{C}(\{q;\bot \mid q \in \mathcal{P}(L) \setminus \{\bot, \top\}\})$. Then $u \subseteq v$ implies v is also a cube of $\{q;\bot \mid q \in \mathcal{P}(L) \setminus \{\bot, \top\}\}$. This implies:
    * Either $c = (p, q;\bot)$, $p \leq \bot;\bot$, $q < \top$. This implies $u = (\vec{\bot}_q;\bot, q;\bot)$. Then, $u \subseteq v$ implies $v = (\vec{\bot}_q;\bot, t;\bot)$, $t \geq q$. We define, $d = (p, t;\bot)$. Then $c \subseteq d$, and $v \in \Theta(\mathcal{C}(X))$ implies $d \in \mathcal{C}(X)$

$*$ or $c = (p; \bot, q)$, $p > \bot$ and $\top; \bot \leq q$, which is dual to the case above.

– $u \in \mathcal{C}(\{\top; q \mid q \in \mathcal{P}(L) \setminus \{\bot, \top\}\})$. Similar to the case above.

□

Lemma 4.4.26. *Given a cover R, and any partition $U \coprod V = R$, such that $U \neq \emptyset$. Then*

$$\coprod_{U_\theta \in S_U} \mathcal{C}(Z_{U_\theta, V_\theta}) = \Theta(\mathcal{C}(Z_{U,V}))$$

With $S_U = \{U_\theta \mid U = \{r \in R \mid U_\theta \cap \Theta(r) \neq \emptyset\}, U_\theta \amalg V_\theta = \Theta(R)\}$

Proof. Let us suppose given a partition $U \amalg V = \Omega(R)$ such that $U \neq \emptyset$. We will proceed by double inclusion:

- Let $c \in \Theta(\mathcal{C}(Z_{U,V}))$ i.e. $c \in \Theta(x), d \in \mathcal{C}(Z_{U,V})$,
  - If $[d] \subseteq [v]^{\mathrm{c}}$, then by Lemma 4.4.23, we have $[c][\Theta(d)] \subseteq [d] \subseteq [v]^{\mathrm{c}} \subseteq [\Theta(v)]^{\mathrm{c}}$
  - $d \subseteq u$ implies by Lemma 4.4.24 that there exists a unique $u_c \in \Theta(u)$ such that $c \subseteq u_c$. Thus, for any $w \in \Theta(u) \setminus u_c$, $[c] \subseteq [w]^{\mathrm{c}}$

  Thus, $d \subseteq \bigcap\limits_{u \in U} [u] \bigcap\limits_{v \in V} [v]^{\mathrm{c}}$ implies

$$[c] \subseteq \bigcap_{u \in U} \left([u_c] \bigcap_{\substack{w \in \Theta(u) \\ w \neq u_c}} [w]^{\mathrm{c}} \right) \bigcap_{v \in V} [\Theta(v)]^{\mathrm{c}}$$

$$[c] \subseteq \bigcap_{u \in U} \left([u_c] \bigcap_{\substack{w \in \Theta(u) \\ w \neq u_c}} [w]^{\mathrm{c}} \right) \bigcap_{v \in V} \bigcap_{w \in \Theta(v)} [w]^{\mathrm{c}}$$

  Then by defining $U_\theta = \{u_c \mid u \in U\}$ and $V_\theta = \Theta(V) \bigcup \Theta(U) \setminus U_\theta$, we have $U_\theta \amalg V_\theta = \Theta(U) \cup \Theta(V) = \Theta(R)$ and

$$c \in \mathcal{C}(Z_{U_\theta, V_\theta})$$

- Given $U \coprod V = R$, let us show that for any $U_\theta \coprod V_\theta = \Theta(R)$ such that $U_\theta \in S_U$, we have

$$Z_{U_\theta, V_\theta} \subseteq Z_{U,V}$$

  for all $u \in U$ such that $U_\theta \cap \Theta(u) \neq \emptyset$, if there exists $u_1 \neq u_2 \in U_\theta \cap \Theta(u)$, then by Lemma 4.4.22, $[u_1] \cap [u_2] = \emptyset$ i.e.

$$Z_{U_\theta, V_\theta} = \emptyset$$

  If that's the case, then we can conclude the whole proof. Let us then suppose that for all $u \in U$ there exists a unique $u_\theta \in \Theta(u)$ such that $u_\theta \in \Theta(u) \cap U_\theta$. Thus, by definition of S_U, we have:

$$U_\theta = \{u_\theta \mid u \in U\} \qquad V_\theta = \bigcup_{u \in U} \{w \in \Theta(u) \mid w \neq u_\theta\} \cup \Theta(V)$$

Then,

$$
\begin{aligned}
Z_{U_\theta,V_\theta} &= \bigcap_{u'\in U_\theta} [u'] \bigcap_{v\in V_\theta} [v]^{\mathrm{c}} \\
&= \bigcap_{u_\theta\in U_\theta} [u_\theta] \bigcap_{u\in U} \Bigg(\bigcap_{\substack{w\in\Theta(u)\\ w\neq u_\theta}} [w]^{\mathrm{c}} \Bigg) \bigcap_{v\in\Theta(V)} [v']^{\mathrm{c}} \\
&= \bigcap_{u\in U} \Bigg([u_\theta] \bigcap_{w\in\Theta(u)\setminus\{u_\theta\}} [w]^{\mathrm{c}} \Bigg) \bigcap_{v\in V} \Bigg(\bigcup_{w\in\Theta(v)} [w] \Bigg)^{\mathrm{c}} \\
&= \bigcap_{u\in U} [u_\theta] \bigcap_{v\in V} [\Theta(v)]^{\mathrm{c}} && \text{Lemma 4.4.22} \\
&\subseteq \bigcap_{u\in U} [u] \bigcap_{v\in V} [v]^{\mathrm{c}} && \text{Lemma 4.4.22} \\
Z_{U_\theta,V_\theta} &\subseteq Z_{U,V}
\end{aligned}
$$

Thus, $\mathcal{C}(Z_{U_\theta,V_\theta}) \subseteq \mathcal{C}(Z_{U,V})$ and hence

$$\Theta(\mathcal{C}(Z_{U_\theta,V_\theta})) \subseteq \Theta(\mathcal{C}(Z_{U,V}))$$

By definition, there exists at least an element $u \in U_\theta$. Thus, $Z_{U_\theta,V_\theta} \subseteq [u]$, which implies by Lemma 4.4.23,

$$\mathcal{C}(Z_{U_\theta,V_\theta}) \subseteq \Theta(\mathcal{C}(Z_{U_\theta,V_\theta})) \subseteq \Theta(\mathcal{C}(Z_{U,V}))$$

Thus, for any cover R and any $U \amalg V = R$ such that $\mathcal{C}(Z_{U,V}) \cap \mathcal{C}_{-1}$, we have

$$\coprod_{U_\theta\in S_U} \mathcal{C}(Z_{U_\theta,V_\theta}) = \Theta(\mathcal{C}(Z_{U,V}))$$

□

Lemma 4.4.27. *Given a cover R and any partition $U \amalg V = R$ such that $U \neq \emptyset$, we have*

$$\coprod_{U_\theta\in S_U} \mathcal{C}^{\max}(Z_{U_\theta,V_\theta}) = \Theta(\mathcal{C}^{\max}(Z_{U,V}))$$

With $S_U = \{U_\theta \mid U = \{r \in R \mid U_\theta \cap \Theta(r) \neq \emptyset\}, U_\theta \amalg V_\theta = \Theta(R)\}$

Proof. By the previous Lemma 4.4.26, we have

$$\coprod_{U_\theta\in X_U} \mathcal{C}(Z_{U_\theta,V_\theta}) = \Theta(\mathcal{C}(Z_{U,V}))$$

Let us prove that the equality holds on maximal cubes by double inclusion

- Let us suppose $c \in \mathcal{C}^{\max}(Z_{U_\theta^0,V_\theta^0})$. By Lemma 4.4.26, we have that $c \in \Theta(i), i \in \mathcal{C}(Z_{U,V})$. Now, let us suppose $i \subseteq j \in \mathcal{C}^{\max}(Z_{U,V})$, Then by Lemma 4.4.24, there exists $d \in \Theta(j) \subseteq \coprod_{U_\theta \in X_U} \mathcal{C}(Z_{U_\theta,V_\theta})$ such that

$$c \subseteq d$$

Furthermore, as the Z_{U_θ,V_θ} are disjoint, this implies $d \in \mathcal{C}(Z_{U_\theta^0,V_\theta^0})$. Thus, by maximality of c, we have $c = d$, which implies $i = j$, i.e.

$$c \in \Theta(\mathcal{C}^{\max}(Z_{U,V}))$$

- Let us suppose $c \in \Theta(\mathcal{C}^{\max}(Z_{U,V}))$ there exists $i \in \mathcal{C}^{\max}(Z_{U,V})$ such that $c \in \Theta(i)$, By Lemma 4.4.26, we have that $c \in \mathcal{C}(Z_{U_\theta,V_\theta})$, $U_\theta \in X_U$. Let us suppose $d \in \mathcal{C}^{\max}(Z_{U_\theta,V_\theta})$, $U_\theta \in S_U$ such that

$$c \subseteq d$$

By Lemma 4.4.26, $d \in \Theta(\mathcal{C}(Z_{U,V}))$. Then by Lemma 4.4.25 there exists $j \in \mathcal{C}(Z_{U,V})$ such that $d \in \Theta(j)$ and $i \subseteq j$. This implies by maximality of i that $i = j$, and by Lemma 4.4.22 that $c = d$ i.e.

$$c \in \mathcal{C}^{\max}(Z_{U_\theta,V_\theta})$$

□

Proposition 4.4.28. *Let $P \in$ **Pgrm**, for any conservative cover R on a program $\mathcal{U}(P)$,*

$$\Theta \circ \Gamma^m(R) = \Gamma^m \circ \Theta(R)$$

Proof. First we will write $X = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} X_U$ with

$$X_U = \{U_\theta \mid U = \{r \in R \mid U_\theta \cap \Theta(r) \neq \emptyset\}, U_\theta \amalg V_\theta = \Theta(R)\}$$

as previously defined. Let us prove that

$$X = \{U_\theta \mid U_\theta \neq \emptyset, U_\theta \subseteq \Theta(R)\}$$

By definition $X \subseteq \{U_\theta \mid U_\theta \subseteq \Theta(R)\}$. Furthermore, given a partition $U_\theta \amalg V_\theta$, if we define

$$U = \{r \in R \mid U_\theta \cap \Theta(r) \neq \emptyset\} \qquad V = \{r \in R \mid U_\theta \cap \Theta(r) = \emptyset\}$$

We have, $U_\theta \in X_U$ and $U \amalg V = R$, such that $\{U_\theta \mid U_\theta \subseteq \Theta(R)\} \subseteq X$. And thus,

$$X = \{U_\theta \mid U_\theta \neq \emptyset, U_\theta \subseteq \Theta(R)\} \tag{4.4}$$

Now let us proceed by induction on the program P.

- $P \in \mathbf{Prcs}^*$. Given a cover R on P, we get

$$\begin{aligned}
\Theta \circ \Gamma^m(R) &= \bigcup_{\substack{U \bigsqcup V = R \\ U \neq \emptyset}} \Theta(\mathcal{C}^{\max}(Z_{U,V})) \\
&= \bigcup_{\substack{U \bigsqcup V = R \\ U \neq \emptyset}} \coprod_{U_\theta \in X_U} \mathcal{C}^{\max}(Z_{U_\theta, V_\theta}) && \text{Lemma 4.4.27} \\
&= \bigcup_{\substack{U_\theta \bigsqcup V_\theta = \Theta(R) \\ U_\theta \neq \emptyset}} \mathcal{C}^{\max}(Z_{U_\theta, V_\theta}) && \text{Eq. (4.4)} \\
\Theta \circ \Gamma^m(R) &= \Gamma^m \circ \Theta(R)
\end{aligned}$$

- $P = P_1 \| \cdots \| P_n$, for all $1 \leq i \leq n$, $P_i \in \mathbf{Prcs}^*$. Then for any cover S on P, we have

$$\Gamma^m(S) = \left(\prod_{1 \leq k \leq n} \Gamma^m(\mathcal{P}_k(S)) \right) \bigcap \mathcal{C}[S]$$

  where $\mathcal{P}_k(S) = \{c_k \mid c_1 \| \ldots \| c_n \in S\}$. We can remark that by linearity

$$\Theta(\mathcal{P}_k(S)) = \mathcal{P}_k(\Theta(S))$$

  We have for any $1 \leq k \leq n$,

$$\begin{aligned}
\Theta \circ \Gamma^m(\mathcal{P}_k(R)) &= \Gamma^m(\Theta(\mathcal{P}_k(R))) && \text{by the case above} \\
\Theta \circ \Gamma^m(\mathcal{P}_k(R)) &= \Gamma^m(\mathcal{P}_k(\Theta(R)))
\end{aligned}$$

  Furthermore, by Lemma 4.4.22

$$\mathcal{C}(R) = \mathcal{C}(\Theta(R))$$

  This is true for all $1 \leq k \leq n$, then as Θ distributes over products, we get:

$$\begin{aligned}
\Theta \circ \Gamma^m(R) &= \left(\prod_{1 \leq k \leq n} \Theta \circ \Gamma^m(\mathcal{P}_k(R)) \right) \bigcap \mathcal{C}(R) \\
&= \left(\prod_{1 \leq k \leq n} \Gamma^m\big(\mathcal{P}_k(\Theta(R))\big) \right) \bigcap \mathcal{C}(\Theta(R)) \\
\Theta \circ \Gamma^m(R) &= \Gamma^m \circ \Theta(R)
\end{aligned}$$

□

Proposition 4.4.29. *Let $P \in \mathbf{Pgrm}$, for any conservative cover R on a program $\mathcal{U}(P)$,*

$$\Theta \circ \Gamma^*(R) = \Gamma^* \circ \Theta(R)$$

Proof.

$$\begin{aligned}
\Theta \circ \Gamma^*(R) &= \Theta \circ \Gamma^m \circ \Gamma^m(R) && \text{Proposition 3.3.22} \\
&= \Gamma^m \circ \Theta \circ \Gamma^m(R) && \text{Proposition 4.4.28} \\
&= \Gamma^m \circ \Gamma^m \circ \Theta(R) && \text{Proposition 4.4.28} \\
\Theta \circ \Gamma^*(R) &= \Gamma^* \circ \Theta(R)
\end{aligned}$$

□

4.4.2.3 $\Gamma^* \circ \Theta \circ \Psi^{-1} = \Psi^{-1} \circ \Gamma^* \circ \Omega$

In this part we prove the following commutation

Proposition 4.4.37. *Let $P \in \mathbf{Pgrm}$, R a maximal conservative cover on P. Then*

$$\Psi^{-1} \circ \Gamma^* \circ \Omega(R) = \Gamma^* \circ \Theta \circ \Psi^{-1}(R)$$

We begin by a few technical lemmas, which are for most direct consequences of our definition that are restated in this form for practicality.

Lemma 4.4.30. *Given a cube $u \in \Omega(\mathcal{C}(P))$. Then*

- $\Omega \circ \Omega = \Omega$
- *For any subset $X \subseteq [u]$, $\mathcal{C}(X) \in \Omega(\mathcal{C}(P))$*

Proof. Directly by definition of Ω □

Corollary 4.4.31. *Given a cover R and a partition $U \amalg V = \Omega(R)$, $U \neq \emptyset$, then*

$$\mathcal{C}(Z_{U,V}) = \Omega(\mathcal{C}(Z_{U,V}))$$

Proof. As $U \neq \emptyset$, there exists $u \in \Omega(R)$ such that $Z_{U,V} \subseteq [u]$. Thus, by Lemma 4.4.30 $\mathcal{C}(Z_{U,V}) \in \Omega(\mathcal{C}(P))$, and $\mathcal{C}(Z_{U,V}) = \Omega(\mathcal{C}(Z_{U,V}))$ □

We have a complementary version of Proposition 4.3.22, that implies the reflection of order on lifting of cut cubes.

Lemma 4.4.32. *Let $P \in \mathbf{Prcs}$, R a conservative cover of $P^\star$. Let $i, j \in \Psi^{-1}(\Omega(R))$. Then*

$$\Psi(i) \subseteq \Psi(j) \implies \exists k \in \Psi^{-1}(\Psi(j)), i \subseteq k$$

Proof. First let us remark that $\Psi(j) \notin \mathcal{L}_1 \cap \Omega(R)$. As by definition, no cubes of $\Omega(R)$ can contain $\top^\star$, we are left with $\Psi(i), \Psi(j) \in \mathcal{C}_0 \cup \mathcal{L}_5$.

- $\Psi(i) \in \mathcal{C}_0$. Then $\Psi(i) \subseteq \Psi(j)$, $\Psi(j) \in \Omega(R)$ implies $\Psi(i) = \Psi(j)$, and then $i = j$ by Proposition 4.2.36.
- $\Psi(i) \in \mathcal{L}_5$. Then $\Psi(i) \subseteq \Psi(j)$ implies $\Psi(j) \in \mathcal{L}_5$. Then $\Psi(i) = (p^\star, q^\star) \neq (\bot^\star, \bot^\star)$. Then $\Psi(i) = (p^\star, q^\star) \subseteq (x^\star, y^\star) = j$ implies by Lemma 4.3.16, $(p, q) \subseteq (x, y)$ and by induction rules

  $$(p;\bot, q;\bot) \subseteq (x;\bot, y;\bot) \approx j$$
  $$(\top;p, \top;q) \subseteq (\top;x, \top;y) \approx j$$

  As $i \in \{(p;\bot, q;\bot), (\top;p, \top;q)\}$ by definition, this concludes the case.

□

Lemma 4.4.33. *Let $P \in \mathbf{Pgrm}$, R a conservative cover on P. Let $c, d \in \Omega(R)$. Then*

- $[c] \subseteq [d]^{\mathrm{c}}$ *implies for all $x \in \Psi^{-1}(c)$, $[x] \subseteq \bigcap_{y \in \Psi^{-1}(d)} [y]^{\mathrm{c}}$*

Proof. Let $P \in \mathbf{Pgrm}$, R a conservative cover on P. Let $c, d \in \Omega(R)$.

- By contraposition, let us suppose that there exists $x \in \Psi^{-1}(c)$, $y \in \Psi^{-1}(d)$ such that

$$\begin{aligned} & [x] \subseteq [y] \\ \implies & \Phi[x] \subseteq \Phi[y] \\ \implies & [\Psi(x)] \subseteq [\Psi(y)] && \text{Proposition 4.2.44} \\ \implies & [c] \subseteq [d] \\ \implies & [c] \not\subseteq [d]^{\mathrm{c}} \end{aligned}$$

  Thus $[c] \subseteq [d]^{\mathrm{c}}$ implies for all $x \in \Psi^{-1}(c)$, $[x] \subseteq \bigcap\limits_{y \in \Psi^{-1}(d)} [y]^{\mathrm{c}}$

□

The core of the commutation is proven in the following lemmas by proving that the maximal cover of the associated partition supports commute with Ψ^{-1}

Lemma 4.4.34. *Given a cover R on a program $P \in \mathbf{Prcs}^*$, such that $R = \Omega(R)$ and any partition $U \amalg V = R$.*

$$\coprod_{U_\psi \in X_U} \mathcal{C}(Z_{U_\psi, V_\psi}) = \Psi^{-1}(\mathcal{C}(Z_{U,V}))$$

With $X_U = \{U_\psi \mid U = \{r \in R \mid U_\psi \cap \Psi^{-1}(r) \neq \emptyset\}, U_\psi \amalg V_\psi = \Psi^{-1}(R)\}$

Proof. Let us suppose given a partition $U \amalg V = \Omega(R)$, $U \neq \emptyset$. We will proceed by double inclusion:

- Let $c \in \Psi^{-1}(\mathcal{C}(Z_{U,V}))$ i.e. $c \in \Psi^{-1}(x), x \in \mathcal{C}(Z_{U,V}) = \Omega(\mathcal{C}(Z_{U,V}))$ (by Corollary 4.4.31)
  - By Lemma 4.4.33 $[x] \subseteq [v]^{\mathrm{c}}$ implies $[c] \subseteq [\Psi^{-1}(v)]^{\mathrm{c}}$
  - $x \subseteq u$ implies by Lemma 4.4.32 that there exists $u_c \in \Psi^{-1}(u)$ such that $c \subseteq u_c$. And, by Proposition 4.2.40, for any $w \in \Psi^{-1}(u)$, $w \neq u_c$ implies $[c] \subseteq [w]^{\mathrm{c}}$

  Thus, $x \subseteq \bigcap\limits_{u \in U} [u] \bigcap\limits_{v \in V} [v]^{\mathrm{c}}$ implies

$$[c] \subseteq \bigcap_{u \in U} \left([u_c] \bigcap_{\substack{w \in \Psi^{-1}(u) \\ w \neq u_c}} [w]^{\mathrm{c}} \right) \bigcap_{v \in V} [\Psi^{-1}(v)]^{\mathrm{c}}$$

$$[c] \subseteq \bigcap_{u \in U} \left([u_c] \bigcap_{\substack{w \in \Psi^{-1}(u) \\ w \neq u_c}} [w]^{\mathrm{c}} \right) \bigcap_{v \in V} \bigcap_{w \in \Psi^{-1}(v)} [w]^{\mathrm{c}}$$

  Then by defining $U_\psi = \{u_c \mid u \in U\}$ and $V_\psi = \Psi^{-1}(V) \bigcup \Psi^{-1}(U) \setminus U_\psi$, we have $U_\psi \amalg V_\psi = \Psi^{-1}(U) \cup \Psi^{-1}(V) = \Psi^{-1}(R)$ and

$$c \in \mathcal{C}(Z_{u_\psi, V_\psi})$$

- Given a partition $U \amalg V = \Omega(R)$. Let $c \in \mathcal{C}(Z_{U_\psi, V_\psi})$ such that $U_\psi \in X_U$.
  - By Proposition 4.3.18, $c \subseteq u_\psi$ implies $\Psi(c) \subseteq \Psi(u_\psi)$
  - By definition of X_U, we have that for any $v_\psi \in V_\psi$, with $v = \Psi(v_\psi)$, we have that $\Psi^1(v) \subseteq V_\psi$. Thus, for any $v \in V$, $[c] \subseteq [\Psi^{-1}(v)]^{\mathrm{c}}$. By contraposition of Lemma 4.4.32, we get that $[\Psi(c)] \subseteq [\Psi \circ \Psi^{-1}(v)]^{\mathrm{c}} = [v]^{\mathrm{c}}$

  Thus we get

$$\Psi(c) \subseteq \bigcap_{u_\psi \in U_\psi} [\Psi(u_\psi)] \bigcap_{v \in V} [v]$$
$$\Psi(c) \subseteq \bigcap_{u \in U} [u] \bigcap_{v \in V} [v]$$

  i.e. $c \in \Psi^{-1}(\mathcal{C}(Z_{U,V}))$

□

Lemma 4.4.35. *Given a cover R, such that $R = \Omega(R)$ and any $U \amalg V = R$ such that $U \neq \emptyset$*

$$\coprod_{U_\psi \in X_U} \mathcal{C}^{\max}(Z_{U_\psi, V_\psi}) = \Psi^{-1}(\mathcal{C}^{\max}(Z_{U,V}))$$

With $X_U = \{U_\psi \mid U = \{r \in R \mid U_\psi \cap \Psi^{-1}(r) \neq \emptyset\}, U_\psi \amalg V_\psi = \Psi^{-1}(R)\}$

Proof. By the previous Lemma 4.4.34, we have

$$\coprod_{U_\psi \in X_U} \mathcal{C}(Z_{U_\psi, V_\psi}) = \Psi^{-1}(\mathcal{C}(Z_{U,V}))$$

Let us prove that the equality holds on maximal cubes by double inclusion

- Let us suppose $c \in \mathcal{C}^{\max}(Z_{U_\psi^0, V_\psi^0})$. By Lemma 4.4.34, we have that $c \in \Psi^{-1}(x), x \in \mathcal{C}(Z_{U,V})$. Now, let us suppose $x \subseteq y \in \mathcal{C}^{\max}(Z_{U,V})$, Then by Lemma 4.4.32, there exists $d \in \Psi^{-1}(y)$ i.e. $d \in \coprod_{U_\psi \in X_U} \mathcal{C}(Z_{U_\psi, V_\psi})$ such that

$$c \subseteq d$$

  As the Z_{U_ψ, V_ψ} are disjoint, this implies $d \in \mathcal{C}^{\max}(Z_{U_\psi^0, V_\psi^0})$. Thus, $c = d$, which implies $x = y$, i.e.

$$c \in \Psi^{-1}(\mathcal{C}^{\max}(Z_{U,V}))$$

- Let us suppose $c \in \Psi^{-1}(\mathcal{C}^{\max}(Z_{U,V}))$. By Lemma 4.4.34, we have that $c \in \mathcal{C}(Z_{U_\psi, V_\psi})$, $U_\psi \in X_U$. Let us suppose $d \in \mathcal{C}^{\max}(Z_{U_\psi, V_\psi})$ such that

$$c \subseteq d$$

  Then by Proposition 4.3.18 $\Psi(c) \subseteq \Psi(d)$. This implies by maximality of $\Psi(c)$ that $\Psi(c) = \Psi(d)$ and by Proposition 4.2.40, $c \subseteq d$ implies $c = d$.

□

Proposition 4.4.36. *Let $P \in \mathbf{Pgrm}$, R a maximal conservative cover on P. Then*

$$\Psi^{-1} \circ \Gamma^m(\Omega(R)) = \Gamma^m \circ \Theta \circ \Psi^{-1}(R)$$

Proof. First we will write $X = \bigcup_{\substack{U \amalg V = R \\ U \neq \emptyset}} X_U$ with

$$X_U = \{U_\psi \mid U = \{r \in R \mid U_\psi \cap \Psi^{-1}(r) \neq \emptyset\}, U_\psi \amalg V_\psi = \Psi^{-1}(R)\}$$

as previously defined. Let us prove that

$$X = \{U_\psi \mid U_\psi \neq \emptyset, U_\psi \subseteq \Psi^{-1}(R)\}$$

By definition $X \subseteq \{U_\psi \mid U_\psi \subseteq \Psi^{-1}(R)\}$. Furthermore, given a partition $U_\psi \amalg V_\psi =$, $U_\psi \neq \emptyset$, if we define

$$U = \{r \in R \mid U_\psi \cap \Psi^{-1}(r) \neq \emptyset\} \qquad V = \{r \in R \mid U_\psi \cap \Psi^{-1}(r) = \emptyset\}$$

We have, $U \neq \emptyset$ as $\mathcal{U}_\psi \neq \emptyset$. Thus, $U_\psi \in X_U$ and $U \amalg V = R$, such that $\{U_\psi \mid U_\psi \subseteq \Psi^{-1}(R)\} \subseteq X$. And thus,

$$X = \{U_\psi \mid U_\psi \neq \emptyset, U_\psi \subseteq \Psi^{-1}(R)\} \tag{4.5}$$

Now we will proceed by induction on P

- $P \in \mathbf{Prcs}^*$. Given R a cover on P, such that $R = \bigcup_{i \in I} R_i$, with R_i maximal conservative. As $\Omega(R) = \Omega(\Omega(R))$, we can apply the previous lemmas

$$\begin{aligned}
\Psi^{-1} \circ \Gamma^m \circ \Omega(R) &= \bigcup_{\substack{U \sqcup V = \Omega(R) \\ U \neq \emptyset}} \Psi^{-1}(\mathcal{C}^{\max}(Z_{U,V})) \\
&= \bigcup_{\substack{U \sqcup V = \Omega(R) \\ U \neq \emptyset}} \coprod_{U_\psi \in X_U} \mathcal{C}^{\max}(Z_{U_\psi, V_\psi}) && \text{Lemma 4.4.35} \\
&= \bigcup_{\substack{U_\psi \sqcup V_\psi = \Psi^{-1} \circ \Omega(R) \\ U_\psi \neq \emptyset}} \mathcal{C}^{\max}(Z_{U_\psi, V_\psi}) && \text{Eq. (4.5)} \\
&= \Gamma^m \circ \Psi^{-1} \circ \Omega(R) \\
&= \Gamma^m \circ \Psi^{-1} \circ \Omega(\bigcup_{i \in I} R_i) \\
&= \Gamma^m \circ \bigcup_{i \in I} \Psi^{-1} \circ \Omega(R_i) \\
&= \Gamma^m \circ \bigcup_{i \in I} \Theta \circ \Psi^{-1}(R_i) && \text{Proposition 4.4.21} \\
\Psi^{-1} \circ \Gamma^m \circ \Omega(R) &= \Gamma^m \circ \Theta \circ \Psi^{-1}(R)
\end{aligned}$$

- $P = P_1 \| \cdots \| P_n$, $P_k \in \mathbf{Prcs}^*$. Then for any cover S on P, we have

$$\Gamma^m(S) = \Big(\prod_{1 \leq k \leq n} \Gamma^m(\mathcal{P}_k(S)) \Big) \bigcap \mathcal{C}[S]$$

where $\mathcal{P}_k(S) = \{c_k \mid c_1 \| \ldots \| c_n \in S\}$. We can remark that by linearity

$$\mathcal{P}_k(\Omega(S)) = \Omega(\mathcal{P}_k(S)) \quad \Psi^{-1}(\mathcal{P}_k(S)) = \mathcal{P}_k(\Psi^{-1}(S)) \quad \mathcal{P}_k(\Theta(S)) = \Theta(\mathcal{P}_k(S))$$

As the case of sequential process above does not require R to be maximal, we have for any $1 \leq k \leq n$,

$$\begin{aligned} \Psi^{-1} \circ \Gamma^m(\mathcal{P}_k(\Omega(R))) &= \Psi^{-1} \circ \Gamma^m \circ \Omega(\mathcal{P}_k(R)) \\ &= \Gamma^m \circ \Theta \circ \Psi^{-1}(\mathcal{P}_k(R)) \qquad \text{by the case above} \\ \Psi^{-1} \circ \Gamma^m(\mathcal{P}_k(\Omega(R))) &= \Gamma^m \circ \mathcal{P}_k(\Theta \circ \Psi^{-1}(R)) \end{aligned}$$

As R is maximal and conservative, by Proposition 4.3.26

$$\Psi^{-1}(\mathcal{C}(\Omega(R))) = \Psi^{-1}(\mathcal{C}(R)) = \mathcal{C}(\Psi^{-1}(R)) \setminus \mathcal{C}_{-1} = \mathcal{C}(\Theta \circ \Psi^{-1}(R)) \setminus \mathcal{C}_{-1}$$

This is true for all $1 \leq k \leq n$, then as Ψ^{-1} and Ω distribute over products, we get:

$$\begin{aligned} \Psi^{-1} \circ \Gamma^m \circ \Omega(R) &= \Big(\prod_{1 \leq k \leq n} \Psi^{-1} \circ \Gamma^m(\mathcal{P}_k(\Omega(R))) \Big) \bigcap \Psi^{-1}(\mathcal{C}(\Omega(R))) \\ &= \Big(\prod_{1 \leq k \leq n} \Gamma^m\big(\mathcal{P}_k(\Theta \circ \Psi^{-1}(R))\big) \Big) \bigcap \mathcal{C}(\Theta \circ \Psi^{-1}(R)) \setminus \Pi_{\mathcal{U}(P)} \mathcal{C}_{-1} \\ \Psi^{-1} \circ \Gamma^m \circ \Omega(R) &= \Gamma^m \circ \Theta \circ \Psi^{-1}(R) \end{aligned}$$

As by definition, no cube of $\prod_{1 \leq k \leq n} \Gamma^m\big(\mathcal{P}_k(\Theta \circ \Psi^{-1}(R))\big)$ are in $\Pi_{\mathcal{U}(P)} \mathcal{C}_{-1}$ □

Proposition 4.4.37. *Let $P \in \mathbf{Pgrm}$, R a maximal conservative cover on P. Then*

$$\Psi^{-1} \circ \Gamma^* \circ \Omega(R) = \Gamma^* \circ \Theta \circ \Psi^{-1}(R)$$

Proof.

$$\begin{aligned}
\Psi^{-1}\circ\Gamma^*(\Omega(R)) &= \Psi^{-1}\circ\Gamma^m\circ\Gamma^m(\Omega(R)) && \text{Lemma 3.3.20}\\
&= \Psi^{-1}\coprod_{\substack{U\amalg V=\Omega(R)\\U\neq\emptyset}}\Gamma^m(\mathcal{C}^{\max}(Z_{U,V})) && \text{Lemma 3.3.21}\\
&= \coprod_{\substack{U\amalg V=\Omega(R)\\U\neq\emptyset}}\Psi^{-1}\circ\Gamma^m(\mathcal{C}^{\max}(Z_{U,V}))\\
&= \coprod_{\substack{U\amalg V=\Omega(R)\\U\neq\emptyset}}\Psi^{-1}\circ\Gamma^m\circ\Omega(\mathcal{C}^{\max}(Z_{U,V})) && \text{Corollary 4.4.31}\\
&= \coprod_{\substack{U\amalg V=\Omega(R)\\U\neq\emptyset}}\Gamma^m\circ\Theta\circ\Psi^{-1}(\mathcal{C}^{\max}(Z_{U,V})) && \text{Proposition 4.4.36}\\
&= \Gamma^m\Big(\coprod_{\substack{U\amalg V=\Omega(R)\\U\neq\emptyset}}\Theta\circ\Psi^{-1}(\mathcal{C}^{\max}(Z_{U,V}))\Big) && \text{by the remark below}\\
&= \Gamma^m\circ\Theta\circ\Psi^{-1}\Big(\coprod_{\substack{U\amalg V=\Omega(R)\\U\neq\emptyset}}\mathcal{C}^{\max}(Z_{U,V})\Big)\\
&= \Gamma^m\circ\Theta\circ\Psi^{-1}\circ\Gamma^m\circ\Omega(R)\\
&= \Gamma^m\circ\Theta\circ\Gamma^m\circ\Theta\Psi^{-1}(R) && \text{Proposition 4.4.36}\\
&= \Gamma^m\circ\Theta\circ\Theta\circ\Gamma^m\circ\Psi^{-1}(R) && \text{Proposition 4.4.28 as } \Psi^{-1}(R) \text{ maximal}\\
&= \Gamma^m\circ\Theta\circ\Gamma^m\circ\Psi^{-1}(R)\\
&= \Gamma^m\circ\Gamma^m\circ\Theta\circ\Psi^{-1}(R) && \text{Proposition 4.4.28}\\
\Psi^{-1}\circ\Gamma^*(\Omega(R)) &= \Gamma^*\circ\Theta\circ\Psi^{-1}(R)
\end{aligned}$$

□

We can now prove the full commutation announced at the start of the section

Proposition 4.4.38. *Given a program $P\in\mathbf{Pgrm}$ and a maximal conservative cover R on P we have*

$$\Theta\circ\Gamma^*\circ\Psi^{-1}(R)=\Psi^{-1}\circ\Gamma^*\Omega(R)$$

Proof. Given a program $P\in\mathbf{Pgrm}$ and a maximal conservative cover R on P we have

$$\begin{aligned}
\Theta\circ\Gamma^*\circ\Psi^{-1}(R) &= \Gamma^*\circ\Theta\circ\circ\Psi^{-1}(R) && \text{Proposition 4.4.29}\\
&= \mathit{ccps}\circ\Psi^{-1}\circ\Omega(R) && \text{Proposition 4.4.21}\\
\Theta\circ\Gamma^*\circ\Psi^{-1}(R) &= \Psi^{-1}\circ\mathit{ccps}\circ\Omega(R) && \text{Proposition 4.4.37}
\end{aligned}$$

□

As Ω and Θ preserve reachability, this could be enough to define Algorithm 4.4.51, but cutting through Ω really is quite costly in terms of number of cubes generated, and we are going to prove that the only cube that Ω needs to cut is the cube $(\bot,\top)$ to comply with the action of Θ.

4.4.2.4 $\Gamma^* \circ \Omega = \Gamma^* \circ \chi$

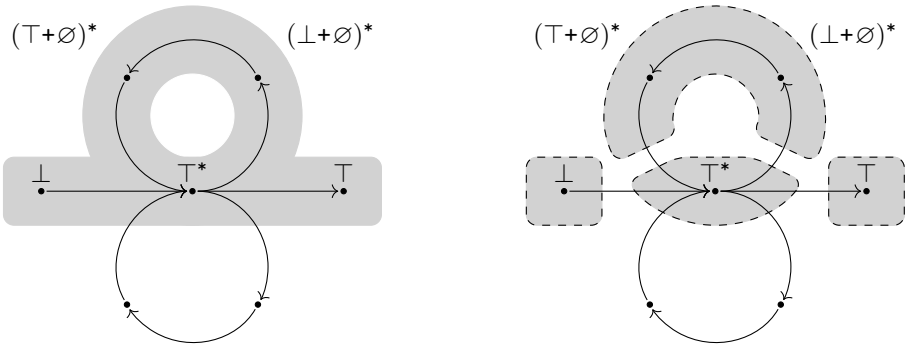
In this section we prove that the operator Ω is entirely unnecessary when considering the partition, and can instead be replaced by χ.

Proposition 4.4.44. *Let $P \in$ **Pgrm**, let R a conservative maximal cover of P. Then*

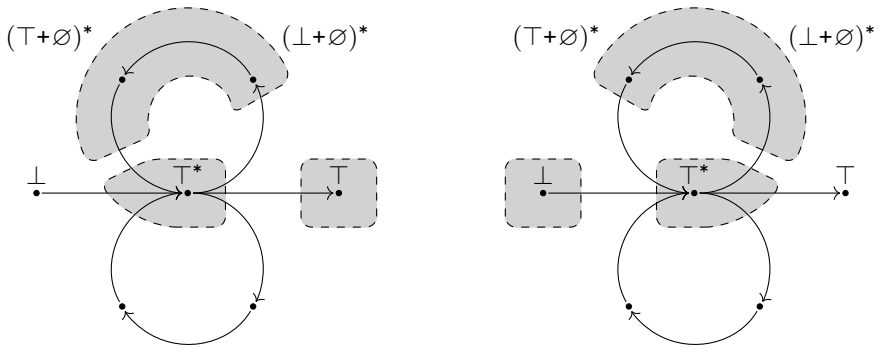
$$\Gamma^*(\Omega(R)) = \Gamma^*(\chi(R))$$

The core of the proof lies in Proposition 4.4.41 which gives an equivalence between partition of $\Omega(R)$ and $\chi(R)$.

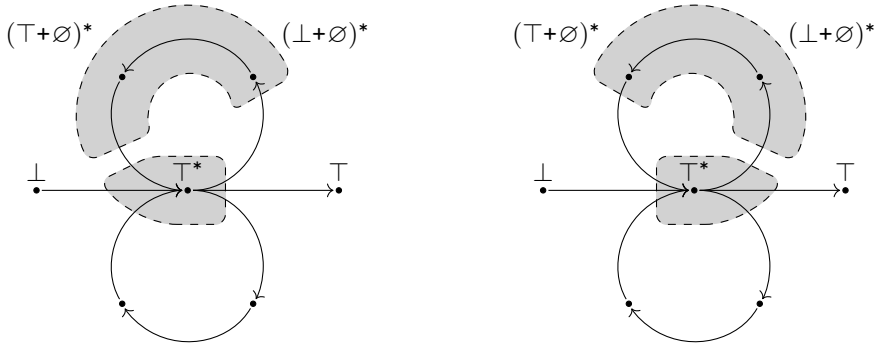
Example 4.4.39. Indeed, by Proposition 4.3.21, the cubes of a maximal conservative cover that cross the core will always be grouped in quadruplets of the form $(p^\star, \top^\star)$,$(\top^\star, q^\star)$, $(p^\star, \top^\star, q^\star)$,$(\bot, \top^\star, \top)$. In the figure below we show the cubical partition associated to such a quadruplet.



Now we give the result of the operations of cutting with Ω.



$\Omega\big((\bot+\varnothing)^\star, \top\big)$ $\Omega\big(\bot, (\top+\varnothing)^\star\big)$



$$\Omega\big((\bot{+}\varnothing)^*, \top^*, (\top{+}\varnothing)^*\big)$$

Now if we apply the cubical partition to the union of these covers, we will obtain exactly the same partition as before. This process generalizes to all maximal conservative covers (Proposition 4.4.41), and all covers that are a union of maximal conservative cover (Corollary 4.4.42). The reason χ appears is that we cut $(\bot, \top)$ whose partition is equal to itself.

Lemma 4.4.40. *Given a program $P \in \mathbf{Pgrm}$ and a cube $c \in \square(\mathcal{U}(P))$, we have*

$$[\Omega(c)] = [c]$$

Proof. By induction on P.

- $P \in \mathbf{Prcs}$. Then Ω is the identity.
- $P = P_1 \,\|\, P_2$. Then, $c \in \square(\mathcal{U}(P))$ implies $c = c_1 \times c_2$ and

$$\begin{aligned} [\Omega(c)] &= [\Omega(c_1)] \times [\Omega(c_2)] \\ &= [c_1] \times [c_2] && \text{induction hypothesis} \\ [\Omega(c)] &= [c] \end{aligned}$$

  Which concludes the proof.

- $P = Q^*$.
  - $c \notin \mathcal{L}_4$. The, by definition of Ω,

$$\begin{aligned} [\Omega(c)] &= [\mathcal{C}^{\max}\big([c] \setminus \{\top^*\}\big) \cup \mathcal{C}^{\max}\big([c] \cap \{\top^*\}\big)] \\ [\Omega(c)] &= [c] \setminus \{\top^*\} \cup ([c] \cap \{\top^*\}) \\ [\Omega(c)] &= [c] \end{aligned}$$

  - $c = (p, \top^*, q^*)$. Then, by definition of Ω,

$$\begin{aligned} [\Omega(c)] &= [\Omega(p, \top^*)] \cup [\Omega(\top^*, q)] \\ &= [p, \top^*] \cup [\top^*, q] && \text{by the case above} \\ &= [(p, \top^*, q^*)] \\ [\Omega(c)] &= [c] \end{aligned}$$

□

Proposition 4.4.41. *Let $P \in \mathbf{Prcs}^*$, $X \subseteq \mathcal{P}(P)$ and R a maximal conservative cover on P. Then, the existence of a partition $U \coprod V = \chi(R)$ such that*

$$X = \bigcap_{u \in U} [u] \bigcap_{v \in V} [v]^{\mathrm{c}}$$

is equivalent to the existence of a partition $U_\Omega \coprod V_\Omega = \Omega(R)$ such that

$$X = \bigcap_{u_\Omega \in U_\Omega} [u_\Omega] \bigcap_{v_\Omega \in V_\Omega} [v_\Omega]^{\mathrm{c}}$$

Proof. For any partition $C \amalg D = S$ of a cover, we write

$$Z_{C,D} = \bigcap_{c \in C} [c] \bigcap_{d \in D} [d]^{\mathrm{c}}$$

Let us proceed by separating the case where our program has a loop.

- $P \in \mathbf{Prcs}$. Then Ω is the identity on cubes. Taking $U^\Omega = U$ and $V^\Omega = V$ suffice.
- $P = Q^*$. First let us remind that $\mathcal{L}_0 \cap R = \emptyset$.
  - $R = \{(\bot, \top)\}$. Then $\Omega(R) = \chi(R)$ which concludes the proof.
  - $\bot^* \notin [R]$. Then by Proposition 4.3.21, there exists $((p_j, q_j))_{j \in J} \in \mathcal{C}(P)^J$, such that for all $j \in J$, $p_j, q_j \notin \{\bot, \top\}$ and

    $$R = \bigcup_{j \in J} \{(p_j^*, q_j^*)\}$$

    By Proposition 4.4.18, this implies $\Omega(R) = R = \chi(R)$ which concludes the proof.
  - $(\bot, \top) \notin R, \top^* \in R$. Then $\chi(R) = R$, thus we simply need to prove the equivalence between partitions of R and $\Omega(R)$.
    By Proposition 4.3.21, there exists $((p_j, q_j))_{j \in J_1 \cup J_2}$, such that for all $j \in J_1 \cup J_2$, we have $p_j, q_j \in \mathcal{P}(P) \setminus \{\bot, \top\}$ and

    $$R = \bigcup_{j \in J_1} \{(p_j^*, q_j^*)\} \bigcup_{j \in J_2} \{(\bot, q_j^*), (p_j^*, \top), (p_j^*, \top^*, q_j^*), (\bot, \top^*, \top)\}$$

    And we write

    $$R_{J_1} = \bigcup_{j \in J_1} \{(p_j^*, q_j^*)\} \qquad R_{J_2} = \bigcup_{j \in J_2} R_j$$

    with for all $j \in J_2$

    $$R_j = \{(\bot, q_j^*), (p_j^*, \top), (p_j^*, \top^*, q_j^*), (\bot, \top^*, \top)\}$$

We define for all $j \in J_2$,

$$\begin{array}{lll} U_1 = R_{J_1} \cap U & U_j = R_j \cap U & U_2 = R_{J_2} \cap U \\ V_1 = R_{J_1} \cap V & V_j = R_j \cap V & V_2 = R_{J_2} \cap V \end{array}$$

By construction we have,

$$\begin{array}{ll} U = U_1 \amalg U_2 & V = V_1 \amalg V_2 \\ U = U_1 \amalg (\bigcup_{j\in J} U_j) & V = V_1 \amalg (\bigcup_{j\in J} V_j) \end{array}$$

Let us prove that for a fixed $j \in J_2$ the existence of the partition $U_j \amalg V_j = R_j$, is equivalent to the existence of a partition $U_j^\Omega \amalg V_j^\omega = \Omega(R_j)$ such that $Z_{U_j,V_j} = ZU_j^\Omega, V_j^\omega$.
Let us fix the notations, we write $p = p_j$, $q = q_j$, and

$$\begin{array}{ll} w = (p^\star, \top) & y = (p^\star, \top^\star, q^\star) \\ x = (\bot, q^\star) & z = (\bot, \top^\star, \top) \end{array}$$

such that $R_j = \{w, x, y, z\}$ By definition,

$$\Omega(R_j) = \Omega\{(p^\star, \top), (\bot, q^\star), (p^\star, \top^\star, q^\star), (\bot, \top^\star, \top)\}$$
$$\Omega(R_j) = \{(p^\star, \overleftarrow{\top}_p{}^*), (\bot^\star, \bot^\star), (\vec{\bot}^*_q, q), (\bot, \bot), (\top, \top)\}$$

* If $U_j = \{x, z\}$ then we have $U_j^\Omega = \{(\bot, \bot)\}$. Indeed,

  $$[p^\star, \overleftarrow{\top}_p{}^*]^c \cap [\bot^\star, \bot^\star]^c \cap [\vec{\bot}^*_q, q]^c \cap [\bot, \bot] \cap [\top, \top]^c = \{\bot\} = [w]^c \cap [x] \cap [y]^c \cap [z]$$

* If $U_j = \{w, z\}$ then we have $U_j^\Omega = \{(\top, \top)\}$. Indeed,

  $$[p^\star, \overleftarrow{\top}_p{}^*]^c \cap [\bot^\star, \bot^\star]^c \cap [\vec{\bot}^*_q, q]^c \cap [\bot, \bot]^c \cap [\top, \top] = \{\top\} = [w] \cap [x]^c \cap [y]^c \cap [z]$$

* If $U_j = \{w, x, y, z\}$ then we have $U_j^\Omega = \{(\bot^\star, \bot^\star)\}$. Indeed,

  $$[p^\star, \overleftarrow{\top}_p{}^*]^c \cap [\bot^\star, \bot^\star] \cap [\vec{\bot}^*_q, q]^c \cap [\bot, \bot]^c \cap [\top, \top]^c = \{\bot^\star\} = [w] \cap [x] \cap [y] \cap [z]$$

* If $U_j = \{w, y\}$ then we have $U_j^\Omega = \{(p^\star, \overleftarrow{\top}_p{}^*)\}$. Indeed, by Proposition 4.4.18

  $$\begin{aligned} [x]^c \cap [y] &= [\bot, \bot]^c \cap [\bot^\star, \bot^\star]^c \cap [\vec{\bot}^*_q, q]^c \cap \left([p^\star, \overleftarrow{\top}_p{}^*] \sqcup [\bot^\star, \bot^\star] \sqcup [\vec{\bot}^*_q, q] \right) \\ &= ([\bot, \bot]^c \cap [\bot^\star, \bot^\star]^c \cap [p^\star, \overleftarrow{\top}_p{}^*]) \cap [\vec{\bot}^*_q, q]^c \\ [x]^c \cap [y] &= [p^\star, \overleftarrow{\top}_p{}^*] \cap [\vec{\bot}^*_q, q]^c \end{aligned}$$

  Furthermore

  $$\begin{aligned} [w] \cap [z]^c &= \left([p^\star, \overleftarrow{\top}_p{}^*] \sqcup [\bot^\star, \bot^\star] \sqcup [\top, \top] \right) \cap \left([\bot, \bot]^c \cap [\bot^\star, \bot^\star]^c \cap [\top, \top]^c \right) \\ &= [p^\star, \overleftarrow{\top}_p{}^*] \cap \left([\bot, \bot]^c \cap [\bot^\star, \bot^\star]^c \cap [\top, \top]^c \right) \\ [w] \cap [z]^c &= [p^\star, \overleftarrow{\top}_p{}^*] \end{aligned}$$

Thus

$$[w] \cap [x]^{\mathrm{c}} \cap [y] \cap [z]^{\mathrm{c}} = [p^\star, \overleftarrow{\top}_p{}^*] \cap [\vec{\bot}_q^*, q]^{\mathrm{c}}$$
$$[w] \cap [x]^{\mathrm{c}} \cap [y] \cap [z]^{\mathrm{c}} = [p^\star, \overleftarrow{\top}_p{}^*] \cap [\bot^\star, \bot^\star]^{\mathrm{c}} \cap [\vec{\bot}_q^*, q]^{\mathrm{c}} \cap [\bot, \bot]^{\mathrm{c}} \cap [\top, \top]^{\mathrm{c}}$$

* Dually, if $U_j = \{x, y\}$ then we have $U_j^\Omega = \{(\vec{\bot}_q^*, q)\}$. Indeed, by the same arguments

$$[w]^{\mathrm{c}} \cap [x] \cap [y] \cap [z]^{\mathrm{c}} = [p^\star, \overleftarrow{\top}_p{}^*]^{\mathrm{c}} \cap [\vec{\bot}_q^*, q]$$
$$[w]^{\mathrm{c}} \cap [x] \cap [y] \cap [z]^{\mathrm{c}} = [p^\star, \overleftarrow{\top}_p{}^*] \cap [\bot^\star, \bot^\star]^{\mathrm{c}} \cap [\vec{\bot}_q^*, q]^{\mathrm{c}} \cap [\bot, \bot]^{\mathrm{c}} \cap [\top, \top]^{\mathrm{c}}$$

* If $U_j = \{w, x, y\}$ then we have $U_j^\Omega = \{(p^\star, \overleftarrow{\top}_p{}^*), (\vec{\bot}_q^*, q)\}$. Indeed,

$$[w] \cap [x] = ([p^\star, \overleftarrow{\top}_p{}^*] \sqcup [\bot^\star, \bot^\star] \sqcup [\top, \top]) \bigsqcap ([\vec{\bot}_q^*, q] \sqcup [\bot^\star, \bot^\star] \sqcup [\bot, \bot])$$
$$[w] \cap [x] = ([p^\star, \overleftarrow{\top}_p{}^*] \cap [\vec{\bot}_q^*, q]) \sqcup [\bot^\star, \bot^\star]$$

By Proposition 4.4.18, $[z] = [\top, \top] \sqcup [\bot^\star, \bot^\star] \sqcup [\bot, \bot]$ i.e. $[z]^{\mathrm{c}} = [\bot^\star, \bot^\star]^{\mathrm{c}} \cap ([\top, \top] \sqcup [\bot, \bot])^{\mathrm{c}}$, such that

$$[\bot^\star, \bot^\star] \cap [z]^{\mathrm{c}} = \emptyset$$

Additionally $[p^\star, \overleftarrow{\top}_p{}^*] \subseteq [z]^{\mathrm{c}}$ and $[\vec{\bot}_q^*, q] \subseteq [z]^{\mathrm{c}}$. Thus,

$$[w] \cap [x] \cap [z]^{\mathrm{c}} = [p^\star, \overleftarrow{\top}_p{}^*] \cap [\vec{\bot}_q^*, q]$$

And once again by Proposition 4.4.18, $[p^\star, \overleftarrow{\top}_p{}^*], [\vec{\bot}_q^*, q] \subseteq [y]$. This implies

$$[w] \cap [x] \cap [z]^{\mathrm{c}} \cap [y] = [p^\star, \overleftarrow{\top}_p{}^*] \cap [\vec{\bot}_q^*, q]$$
$$[w] \cap [x] \cap [z]^{\mathrm{c}} \cap [y] = [p^\star, \overleftarrow{\top}_p{}^*] \cap [\bot^\star, \bot^\star] \cap [\vec{\bot}_q^*, q]^{\mathrm{c}} \cap [\bot, \bot]^{\mathrm{c}} \cap [\top, \top]^{\mathrm{c}}$$

* For all other U_j or U_j^Ω, the intersections of the supports are empty.

Thus, we have for all $j \in J_2$, a partition $U_j^\Omega \amalg V_j^\Omega = \omega(R_j)$ such that

$$Z_{U_j, V_j} = ZU_j^\Omega, V_j^\Omega$$

We define for all $j \in J_2$,

$$U_1^\Omega = \Omega(R_{J_1}) \cap U \qquad U_2 = \bigcup_{j \in J_2} U_j^\Omega \qquad U^\Omega = U_1^\Omega \cup U_2^\Omega$$
$$V_1^\Omega = \Omega(R_{J_1}) \cap V \qquad V_2 = R_{J_2} \cap V \qquad V^\Omega = V_1^\Omega \cup V_2^\Omega$$

By the result above

$$Z_{U_2, V_2} = \bigcap_{j \in J} Z_{U_j, V_j} = \bigcap_{j \in J} Z_{U_j^\Omega, V_j^\Omega} = Z_{U_2^\Omega, V_2^\Omega}$$

By Proposition 4.4.18, this implies $\Omega(R_{J_1}) = R_{J_1} = \chi(R_{J_1})$, i.e. $U_1 = U_1^\Omega$ and $V_1 = V_1^\Omega$, which implies a fortiori

$$Z_{U_1,V_1} = Z_{U_1^\Omega,V_1^\Omega}$$

Furthermore, it also implies $U_1^\Omega \amalg V_1^{\amalg} = \Omega(R_{J_1})$ and by construction $U_2^\Omega \amalg V_2 = \Omega(R_{J_2})$. Thus, we have

$$Z_{U,V} = Z_{U_1,V_1} \cap Z_{U_1,V_1} = Z_{U_1^\Omega,V_1^\Omega} \cap Z_{U_2^\Omega,V_2^\Omega} = Z_{U^\Omega,V^\Omega}$$

This concludes the proof. □

Corollary 4.4.42. *Let $P \in \mathbf{Prcs}^*$, $X \subseteq \mathcal{P}(P)$ and R a cover on P, such that there exists a family $(R_i)_{i\in I}$ of conservative maximal covers such that $R = \bigcup\limits_{i\in I} R_i$. Then, the existence of a partition $U \coprod V = R$ such that*

$$X = \bigcap_{u\in U} [u] \bigcap_{v\in V} [v]^{\mathrm{c}}$$

is equivalent to the existence of a partition $U_\Omega \coprod V_\Omega = \Omega(R)$ such that

$$X = \bigcap_{u_\Omega\in U_\Omega} [u_\Omega] \bigcap_{v_\Omega\in V_\Omega} [v_\Omega]^{\mathrm{c}}$$

Proof. Let us suppose given $U \amalg V = R$. Then if we define

$$U_i = U \cap R_i \qquad\qquad V_i = V \cap R_i$$

Then $U_i \amalg V_i = R_i$ and

$$U = \bigcup_{i\in I} U_i \qquad\qquad V = \bigcup_{i\in I} V_i$$

Then by definition,

$$Z_{U,V} = \bigcap_{i\in I} Z_{U_i,V_i}$$

By Proposition 4.4.41, applied to all $U_i \amalg V_i = R_i$ with R_i conservative maximal, this is equivalent to the existence of a partition $U_i^\Omega \amalg V_i^\omega = \Omega(R_i)$ such that

$$Z_{U_i,V_i} = Z_{U_i^\Omega V_i^\omega}$$

Furthermore, $\Omega(R) = \bigcup\limits_{i\in I} R_i$ implies

$$\bigcup_{i\in I} U_i^\Omega \amalg \bigcup_{i\in I} V_i^\Omega = \Omega(R)$$

Hence by defining $U_\Omega = \bigcup\limits_{i\in I} U_i^\Omega$ and $v_\omega = \bigcup\limits_{i\in I} V_i^\Omega$ we get

$$Z_{U_\Omega,V_\omega} = \bigcap_{i\in I} Z_{U_i^\Omega,V_i^\Omega} = \bigcap_{i\in I} Z_{U_i,V_i} = Z_{U,V}$$

This concludes the proof. □

Proposition 4.4.43. *Let $P \in \mathbf{Pgrm}$, let R a conservative maximal cover of P. Then*

$$\Gamma^m(\Omega(R)) = \Gamma^m(R)$$

Proof. As always, we will need to differentiate $P \in \mathbf{Prcs}^*$ and $P = P_1 \| \dots \| P_n$.

- $P \in \mathbf{Prcs}^*$. Given a cover R on P, such that $R = \bigcup_{i \in I} R_i$ with R_i conservative maximal. We get

$$\begin{aligned}
\Gamma^m(\Omega(R)) &= \bigcup_{U_\Omega \coprod V_\Omega = \Omega(R)} \mathcal{C}^{\max}(Z_{U_\Omega, V_\Omega}) \\
&= \bigcup_{U \coprod V = \chi(R)} \mathcal{C}^{\max}(Z_{U,V}) && \text{Corollary 4.4.42} \\
\Gamma^m(\Omega(R)) &= \Gamma^m(\chi(R))
\end{aligned}$$

- $P = P_1 \| \dots \| P_n$, $P_i \in \mathbf{Prcs}^*$ for all $1 \leq i \leq n$. Then for any cover S on P, we have

$$\Gamma^m(S) = \left(\prod_{1 \leq k \leq n} \Gamma^m(\mathcal{P}_k(S)) \right) \bigcap \mathcal{C}[S]$$

  where $\mathcal{P}_k(S) = \{c_k \mid c_1 \| \dots \| c_n \in S\}$, which is by Corollary 4.3.8 a union of conservative maximal covers. We have for any $1 \leq k \leq n$,

$$\Gamma^m \circ \Omega(\mathcal{P}_k(R)) = \Gamma^m(\chi(\mathcal{P}_k(R))) \qquad \text{by the case above}$$

  This is true for all $1 \leq k \leq n$, then as Ω distributes over products, we get:

$$\begin{aligned}
\Gamma^m \circ \Omega(R) &= \left(\prod_{1 \leq k \leq n} \Gamma^m(\mathcal{P}_k(\Omega(R))) \right) \bigcap \mathcal{C}(\Omega(R)) \\
&= \left(\prod_{1 \leq k \leq n} \Gamma^m(\mathcal{P}_k(\Omega(R))) \right) \bigcap \mathcal{C}(R) \\
&= \left(\prod_{1 \leq k \leq n} \Gamma^m \Omega((\mathcal{P}_k(R))) \right) \bigcap \mathcal{C}(R) \\
&= \left(\prod_{1 \leq k \leq n} \Gamma^m(\mathcal{P}_k(R)) \right) \bigcap \mathcal{C}(\chi(R)) \\
\Theta \circ \Gamma^m(R) &= \Gamma^m(\chi(R))
\end{aligned}$$

□

Proposition 4.4.44. *Let $P \in \mathbf{Pgrm}$, let R a conservative maximal cover of P. Then*

$$\Gamma^*(\Omega(R)) = \Gamma^*(\chi(R))$$

Proof.

$$\begin{aligned} \Gamma^*(\Omega(R)) &= \Gamma^m \circ \Gamma^m(\Omega(R)) && \text{Proposition 3.3.22} \\ &= \Gamma^m \circ \Gamma^m(\chi(R)) && \text{Proposition 4.4.43} \\ \Gamma^*(\Omega(R)) &= \Gamma^*(\chi(R)) && \text{Proposition 3.3.22} \end{aligned}$$

□

4.4.2.5 Final Theorem

Finally, we can prove our main theorem on the commutation of the generic cubical partition operator and the lifting of cubes

Theorem 4.4.45. *Given a program $P \in$ **Pgrm** and a maximal conservative cover R on P we have*

$$\Theta \circ \Gamma^* \circ \Psi^{-1}(R) = \Psi^{-1} \circ \Gamma^*(\chi(R))$$

Proof. By Proposition 4.4.38, we have for any maximal conservative cover

$$\Theta \circ \Gamma^* \circ \Psi^{-1}(R) = \Psi^{-1} \circ \Gamma^*(\Omega(R))$$

By Proposition 4.4.44, this implies

$$\Theta \circ \Gamma^* \circ \Psi^{-1}(R) = \Psi^{-1} \circ \Gamma^*(\chi(R))$$

□

Remark 4.4.46. As $\Psi^{-1}(R^c) = \mathcal{C}^{\max}([R]^c)$ (Theorem 4.3.28) the Theorem 4.4.45 also works when taking the complement.

4.4.3 Deadlock computation for programs with loops

We now have all the tools required to compute unsafe and doomed covers for programs with loops using the previous algorithm of [18] on the finite unfolding of programs. This algorithm is based on a "reachability" order defined on the cubes of the coarsest partition. Once again, we say that a cube R_i is in the past of R_j, when every point of R_i is the origin of a directed path of X that reaches R_j "Being in the past of" is actually the reflexive and transitive closure of the following relation.

Definition 4.4.47. Let $R = (R_i)_{i \in I}$ be a cubical partition of a preorder $\mathcal{P}$. We define the preorder $\lhd$ on elements of R as the following relation: $R_i \lhd R_j$ if and only if for all $x \in [R_i]$ there exists $y \in [R_j]$ and a path of $[R_i] \bigcup [R_j]$ from x to y

Here is the moment where we need to stop seeing $\mathcal{U}(P)$ as a program in and of itself but more as a lens in which we observe its underlying program P. When moving across the cubical partition of $\mathcal{U}(P)$, we should really be thinking of being in both copies of P at the same time. And naturally, when seen in this light, it makes sense to jump from any member of an equivalence class to the other when thinking of "being in one's past".

Definition 4.4.48. Let $P \in \mathbf{Pgrm}$, let R be a conservative cover on P. Let $u, v \in \Gamma^m(\mathcal{C}_P^m \setminus R)$, then we define $\lhd_\approx$ on the quotient of $\Theta_P(\Gamma^m(\mathcal{C}^m_{\mathcal{U}(P)} \setminus \Psi^{-1}(R)))$ by $\approx$ as

$$\Psi^{-1}(u) \lhd_\approx \Psi^{-1}(v) \iff \exists c, d \in \Psi^{-1}(u) \times \Psi^{-1}(v), c \lhd d$$

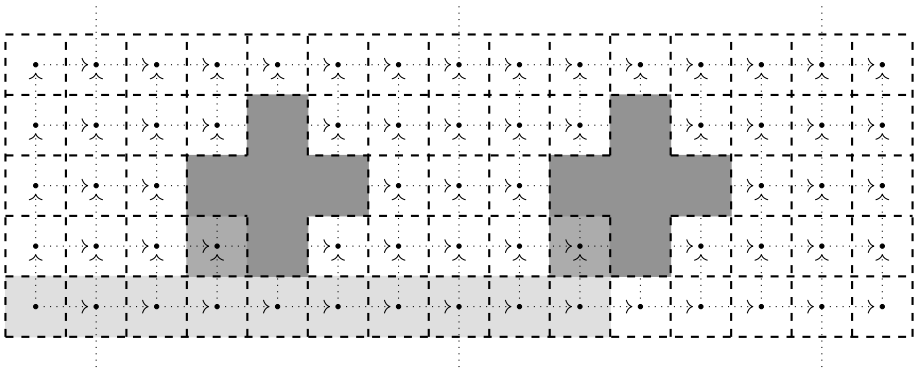
We define $\lhd^*_\approx$ as the reflexive, transitive closure of $\lhd_\approx$

Remark 4.4.49. For conservative programs, loops have no global effect on the consumption of resources, if one can reach one iteration of the loop, the one can reach it in any further iteration of the loop, one simply needs to execute all the parallel loops sequentially, skirting along the edges of the state space.

As an example, let us consider the program $P = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a)^* \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$. Its unfolding is given by the program

$$\mathcal{U}(P) = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a); (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$$

Looking at its generic partition, the two cube in grey are equivalent, and indeed, if the first one can be reached, so can be the second one. Paths to each cube is given by the line of cubes on the lower side of the semantics.



That is why we consider, with our definition, that a cube is, in a sense, both in the past and in the future of any member of its equivalence class.

Now we can finally prove the intuition we have presented at the beginning of this section. Indeed, the following Theorem 4.4.50 tells us precisely that a cube of the looped program is in the past of another if and only if its equivalence class is in the past of the equivalence class of the other.

Theorem 4.4.50. *Let $P \in \mathbf{Pgrm}$, let R be a conservative cover on P. Let $u, v \in \Gamma^m(\mathcal{C}_P^m \setminus R)$, the following properties are equivalent*

- $u \lhd v$
- $\Psi^{-1}(u) \lhd_\approx \Psi^{-1}(v)$
- *There exists $c, d \in \Psi^{-1}(u) \times \Psi^{-1}(v)$ such that $c \lhd d$*

Proof.

- $P \in \mathbf{Prcs}$. As usual Ψ is the identity, so everything works out.

- $P = S \parallel T$. Let $c = s \times t, d = u \times v$ in $\Theta_P(\Gamma^m(\mathcal{C}^m_{\mathcal{U}(P)} \setminus \Psi^{-1}(R)))$ By Definition 4.1.22, a path from $a \parallel b \to p \parallel q$ is a path from $a \to p$ and a path from $b \to q$ such that

$$s \times t \lhd u \times v \iff s \lhd u \text{ and } t \lhd v$$

Similarly

$$\Psi(s \times t) \lhd \Psi(u \times v) \iff \Psi(s) \lhd \Psi(u) \text{ and } \Psi(t) \lhd \Psi(v)$$

Then by applying the induction hypothesis to $\Psi(s) \lhd \Psi(u)$ and $\Psi(t) \lhd \Psi(v)$. This is equivalent to the existence of $s' \approx s, u' \approx u$ and $t' \approx t, v' \approx v$ such that

$$s' \lhd u' \text{ and } t' \lhd v'$$

Thus by the remark above it is equivalent to

$$c = s \times t \approx s' \times t' \lhd u' \times v' \approx u \times v = d$$

- $P = Q^*$. First let us remark that $x \in \Theta_P(\Gamma^m(\mathcal{C}^m_{\mathcal{U}(P)} \setminus \Psi^{-1}(R)))$ implies by definition of Θ
  - $x \in \mathcal{C}_0 \bigcup \mathcal{C}_0$
  - or $x = (s; \bot, t; \bot) \in \mathcal{C}_5$, $\bot < s \leq t < \top$
  - or $x = (\top; s, \top; t)$, $\bot < s \leq t < \top$

  And by Theorem 4.4.45, $x \in \Gamma^m(\mathcal{C}^m_P \setminus R)$ implies $x \in \Psi \circ \Theta_P \circ \Gamma^m(\mathcal{C}^m_{\mathcal{U}(P)} \setminus \Psi^{-1}(R))$ i.e.
  - $x \in \{(\bot, \bot), (\top, \top), (\bot^*, \bot^*)\}$
  - or $x = (s^*, t^*) \in \mathcal{L}_5$ such that $\bot < s \leq t < \top$

  By reduction rules Definition 4.1.22 and by definition of $\mathsf{pred}\cdot$ and $\mathsf{succ}\cdot$ The only reduction rules possible involving $\bot_P, \bot^*, \top_P$ are
  - $\bot \to \bot^*$ and $\bot^* \to \top$
  - $t^* \to \bot^*$, with $t \in \mathsf{pred}\bot$.
  - $\bot^* \to s^*$, with $s \in \mathsf{succ}\bot$.

  Similarly, the only reduction rules involving $\bot_{P;P}, \top_{P;P}, \bot; \bot, \top; \bot, \top; \top$ are
  - $\bot \to \bot; \bot$ and $\top; \bot \to \top$
  - $t; \bot \to \top; \bot$, with $t \in \mathsf{pred}\bot$.
  - $\top; t \to \top; \top$, with $t \in \mathsf{pred}\bot$.
  - $\bot; \bot \to s; \bot$, with $s \in \mathsf{succ}\bot$.
  - $\top; \bot \to \top; s$, with $s \in \mathsf{succ}\bot$.

  Such that the only transition by $\lhd$ involving cubes of $\{(\bot, \bot), (\top, \top), (\bot^*, \bot^*)\}$ and $\mathcal{C}_0 \bigcup \mathcal{C}_0$ are the following (or transition on cubes in the same equivalence class)

- $(\bot^\star, \bot^\star) \lhd (\top, \top)$ and $\Psi^{-1}((\bot^\star, \bot^\star)) \ni (\top;\top, \top;\top) \lhd (\top, \top) \in \Psi^{-1}(\top, \top)$
- $(\bot, \bot) \lhd (\bot^\star, \bot^\star)$ and $\Psi^{-1}(\bot, \bot) \ni (\bot, \bot) \lhd (\bot;\bot, \top;\top) \in \Psi^{-1}((\bot^\star, \bot^\star))$
- $(\bot^\star, \bot^\star) \lhd (s^\star, t^\star)$ with $s \in \mathsf{succ}\bot$ and

$$\Psi^{-1}(\bot^\star, \bot^\star) \ni (\bot;\bot, \bot;\bot) \lhd (s;\bot, t;\bot) \in \Psi^{-1}(s^\star, t^\star)$$

- $(s^\star, t^\star) \lhd (\bot^\star, \bot^\star)$ with $t \in \mathsf{pred}\bot$ and

$$\Psi^{-1}(\bot^\star, \bot^\star) \ni (\bot;\bot, \bot;\bot) \lhd (s;\bot, t;\bot) \in \Psi^{-1}(s^\star, t^\star)$$

Thus for all these cubes

$$u \lhd v \iff \exists c, d \in \Psi^{-1}(u) \times \Psi^{-1}(v), c \lhd d$$

Now let c, d such that $\Psi(c) \lhd \Psi(d)$. The only case left are when $\Psi(c) = (s^\star, t^\star)$ and $\Psi(d) = (u^\star, v^\star)$, such that

$$\Psi^{-1}(\Psi(c)) = \{(s;\bot, t;\bot), (\top;s, \top;t)\}$$
$$\Psi^{-1}(\Psi(d)) = \{(u;\bot, v;\bot), (\top;u, \top;v)\}$$

Then $\Psi(c) \lhd \Psi(d)$ implies for all $x^\star \in [\Psi(c)]$ there exists $y^\star \in [\Psi(d)]$ and a directed path

$$\pi = (z_i^\star)_{i\in I} \colon x^\star \to y^\star \subseteq [\Psi(c)] \cup [\Psi(d)]$$

Then by Lemma 4.2.14 $\top^\star \notin [\Psi(c)] \cup [\Psi(d)]$, i.e. $\top^\star \notin \pi$. Thus, we can apply Definition 4.2.13 which gives $z_i^\star \in [s^\star, t^\star]$ is equivalent to $z_i \in [s, t]$ which is equivalent by reduction rule to

$$z_i;\bot \in [s;\bot, t;\bot]$$

Similarly $z_i^\star \in [u^\star, v^\star]$ is equivalent to

$$z_i;\bot \in [u;\bot, v;\bot]$$

Thus $(z_i;\bot)_{i\in I}$ is a directed path of $[s;\bot, t;\bot] \cup [u;\bot, v;\bot]$ from $x;\bot \in (s;\bot, t;\bot) \approx c$ to $y;\bot \in (u;\bot, v;\bot) \approx d$ Similarly, $(\top;z_i)_{i\in I}$ is a directed path of $[\top;s, \top;t] \cup [\top;u, \top;v]$ from $x;\bot \in (\top;s, \top;t) \approx c$ to $y;\bot \in (\top;u, \top;v) \approx d$

□

Now to compute the unsafe and doomed regions of our programs with loops, we simply need to apply Algorithm 3.3.23 to the partition of the unfolding ordered with $\lhd$ as in Section 1.4.3 and adding arrows in between all elements of a same equivalence class. From this result, applying Ψ one last time will yield the doomed and unsafe regions.

Algorithm 4.4.51. Given a maximal cover R of the forbidden region of the program P.

1. We obtain the maximal cover of the forbidden region of the unfolding as $\Psi^{-1}(R)$. Otherwise, compute $\Phi^{-1}[R]$.
2. We obtain a cover of the maximal cubes of the authorized region in the same way as in Algorithm 3.3.23.

3. Compute the generic partition of the forbidden region as $\Theta \circ \Gamma^m(\Psi^{-1}(R)^c)$.

4. Compute $\lhd_{\approx}$ using Remark 3.3.26.

5. Compute the deadlocks/unsafe/doomed regions using Algorithm 1.4.33 with the altered order

6. The deadlocks of the base programs are obtained by projecting the obtained deadlocks by Φ. Similarly, for the cubes of the unsafe/doomed region we project using Ψ the cubes obtained in the previous step.

An important thing to understand is that the program actually works in the quotient of the unfolding, where each execution is simultaneously in both copies of the loop, that is why the cubes that are equivalent are reachable from each other, we consider that if we have reached one of the cubes, then we can reach the equivalent cube only in a matter of considering a shift in the iteration we are looking at. This of course only works as we are not considering any variables or boolean test for the programs. This method might be possible to adapt to still work with the addition of these feature, but it would require some efforts.

As an addition, our work not only produces the Algorithm 4.4.51 but also transmit an equivalence between the cubical partition, that we hope in later works to carry further, maybe to an eventual notion of category of components from Chapter 2.

Unfortunately visualization of this algorithm is quite complicated, as the simplest of programs with deadlocks and loops necessitates 3D graphs (at least 2 dimensions for the parallel compositions and one for the loops) which makes it very hard to actually represent what is going on in a "real" conservative programs. An attempt has been made in Example 4.4.52. This method is actually implemented in the tool Sparkling [42] made in collaboration with Samuel Mimram, through the form of textual output.

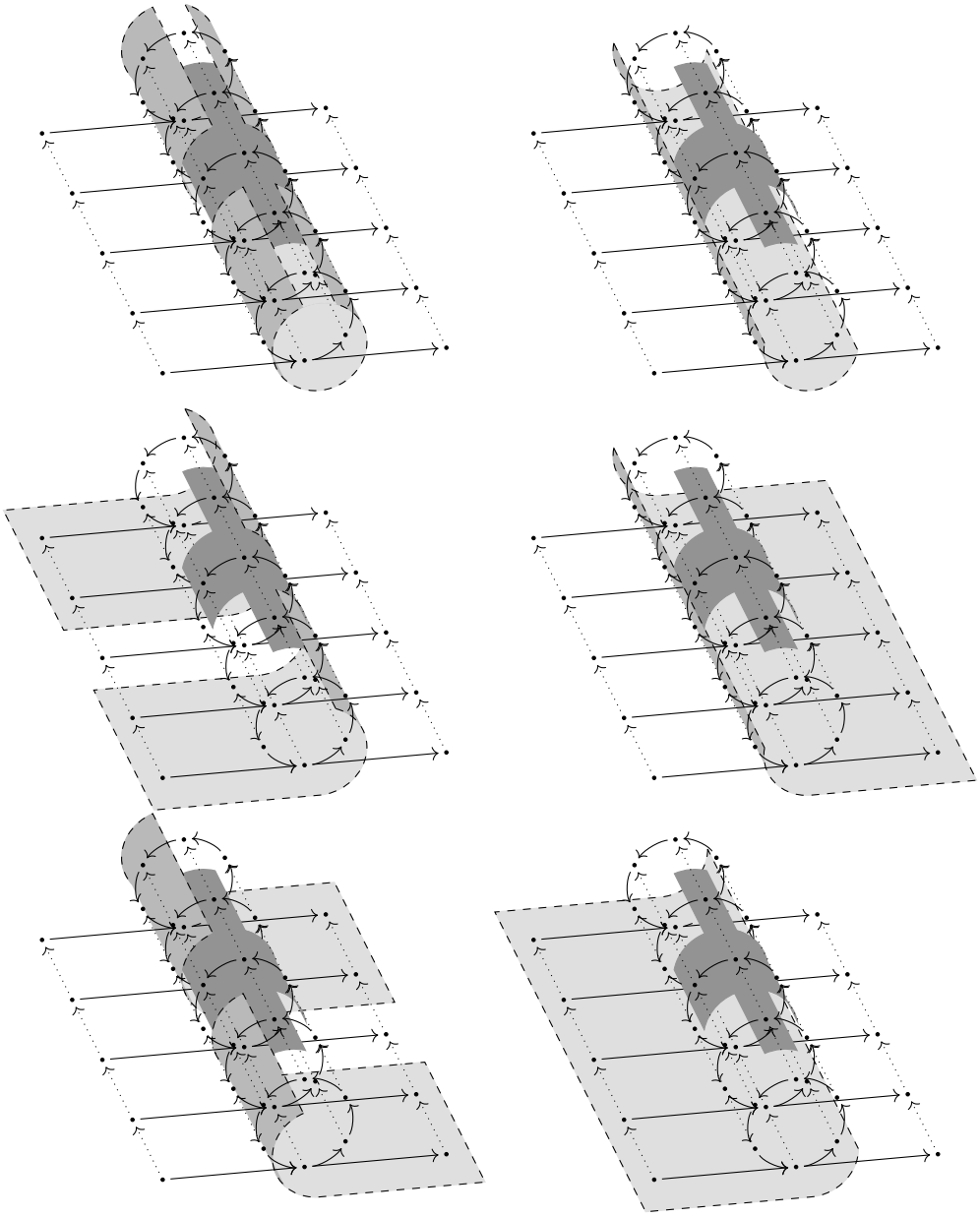
Example 4.4.52. Let us consider the program

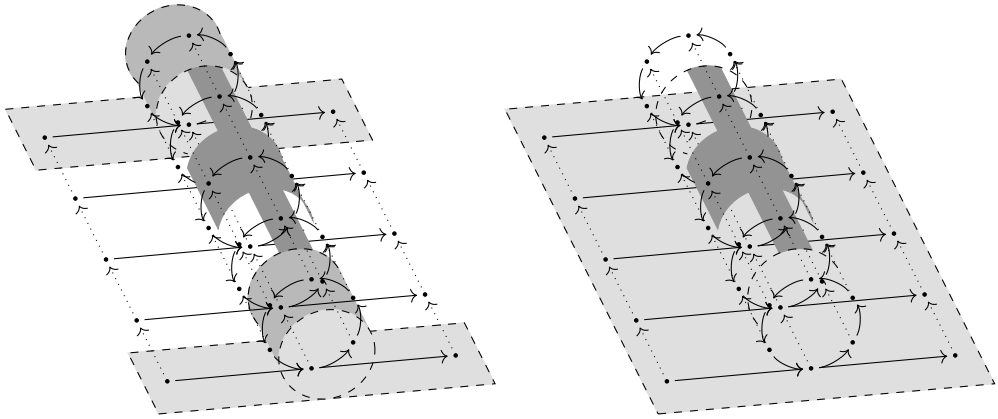
$$P = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a)^* \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$$

Its unfolding is given by the program

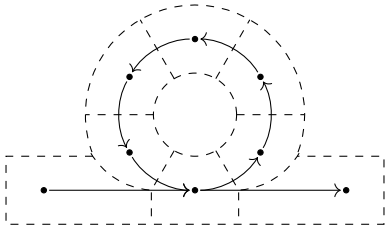
$$\mathcal{U}(P) = (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a); (\mathtt{P}_a; \mathtt{P}_b; \mathtt{V}_b; \mathtt{V}_a) \| (\mathtt{P}_b; \mathtt{P}_a; \mathtt{V}_a; \mathtt{V}_b)$$

Now let us try to see what our algorithm would give on such a program. To make the illustration more clear some transitions have been omitted. They do not change the fundamental results. First and foremost, if we compute the maximal cubes of P we obtain the following cubes.

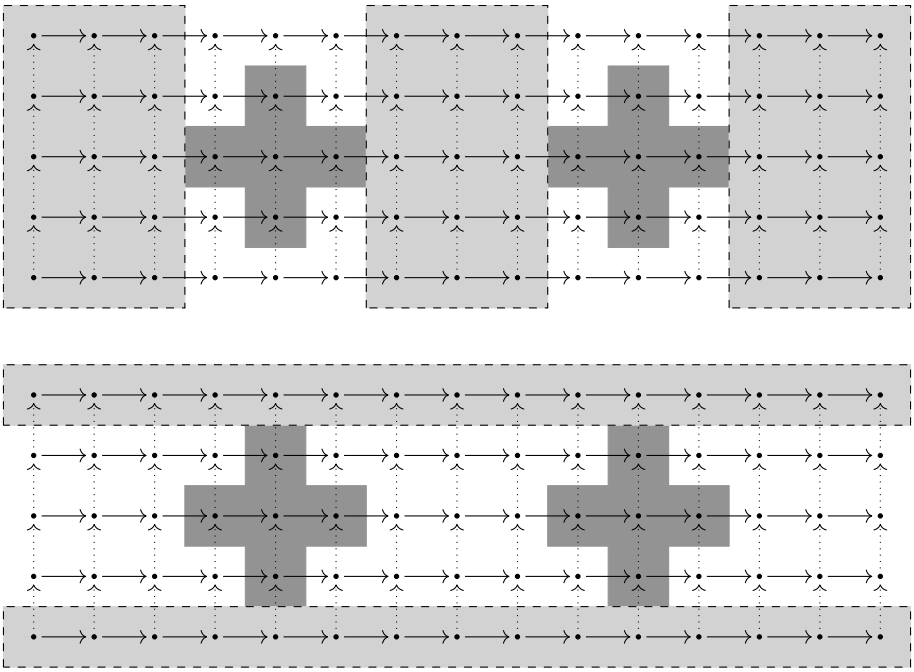


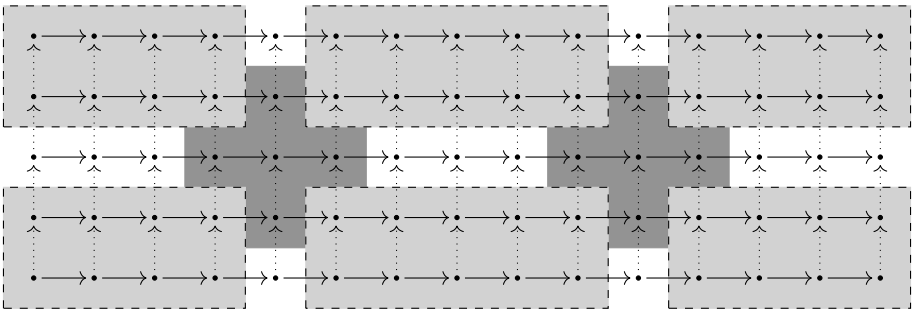


One can see that the generic partition on the base program will separate each position in its own interval. Indeed, if we look at the projection on $(\mathtt{P}_a;\mathtt{P}_b;\mathtt{V}_b;\mathtt{V}_a)^*$ we obtain the following partition.

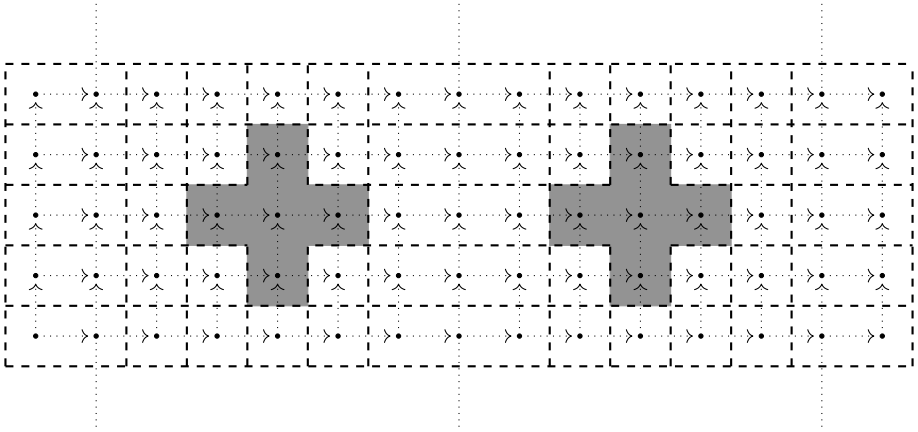


Now if we look at the unfolding of the maximal cubes, we get the following cubes.

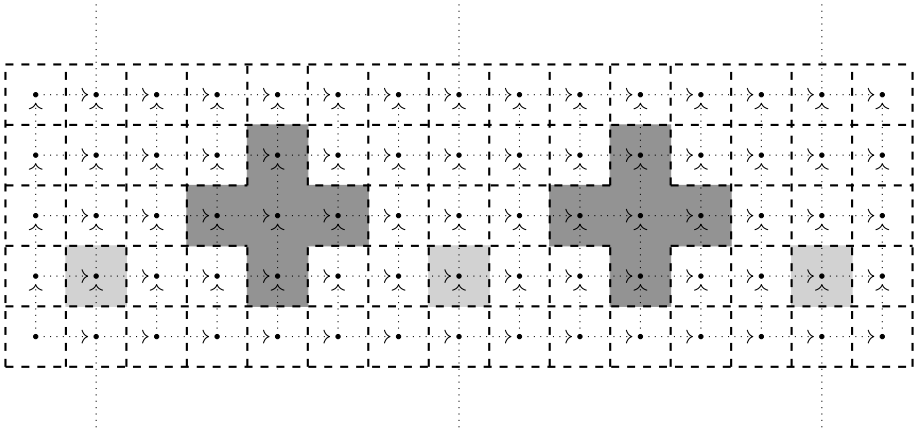




Now, if we apply the generic cubical partition, we get the following cubes.



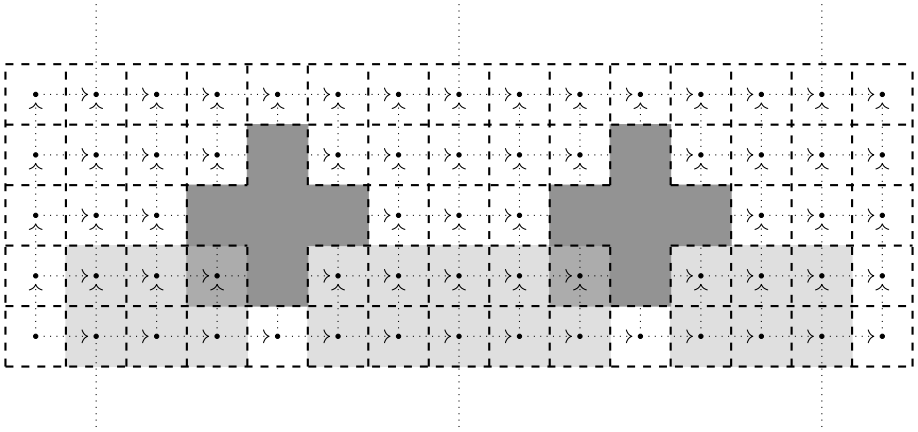
Then, applying Θ, we obtain the following partition.



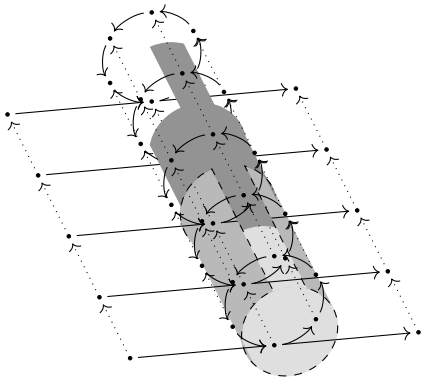
We recall that the order is different from Algorithm 3.3.23. The two cubes in grey are equivalent for $\lhd^*_{\approx}$, meaning we can jump from one to the other, which is the case for any other cube that projects to the same cube on the base program.

The deadlock are easily obtained as the cubes with no transition other than equivalences. Now if we compute the downwards closure of our deadlocks, only taking a single

deadlock would give us the "correct" unsafe region to project, using the aforementioned equivalence of cubes.



Finally, projecting back we get the correct deadlocks and unsafe region for our program with loops.



Remark 4.4.53. In the examples we have that the generic partition separates all positions into their own cubes. This is an artefact due to the fact that we are identifying a lot of positions that would get grouped in the same cube for the sake of presentation.

Chapter 5
Perspectives

Scientists have calculated that the chances of something so patently absurd actually existing are millions to one. But magicians have calculated that million-to-one chances crop up nine times out of ten.

– Sir Terry Pratchett, *Mort*

In this thesis we have provided an extension of Hashimoto's theorem about refining product decompositions of partial orders to the case of loop-free categories. Many of the useful topological invariants associated to loop-free programs are naturally expressed in this larger setting. The factorization properties associated to this theorem can now be used for the purpose of studying independent processes in concurrent programs.

We have also provided a new model of programs, based directly on the syntax of programs, which imports powerful theoretical results from the directed topological models, such as Algorithm 1.4.33, in a setting where implementation is much more natural.

We also provide an extension of the aforementioned algorithm in the case of simple programs with non-nested loops, which offers greater theoretical complexity than the previously known Algorithm 4.1.6 on programs with loops for the computation of doomed and unsafe region, while also being easier to implement per our new setting.

Extension of the deadlock detection algorithm to more general programs

Extending Algorithm 3.3.23 to more than simple programs of **NPIMP** only requires two small steps to be effectively implemented.

The first would be to extend the definition of the generic partition Γ^* to handle programs whose semantics is not defined as a product. This could be done by calculating these partitions locally for each n-dimensional product of semantics and gluing them at the lower dimension cubes at the frontier. The algorithm could then, through iterated computation partition the space, starting from the most nested parallel composition and moving upwards from there.

The second step required would be to give an inductive characterization of cubes, more particularly maximal cubes, in order to perform the separation. Indeed, for now, the characterization of the cubes of a specific constructor, especially in the case of cubes

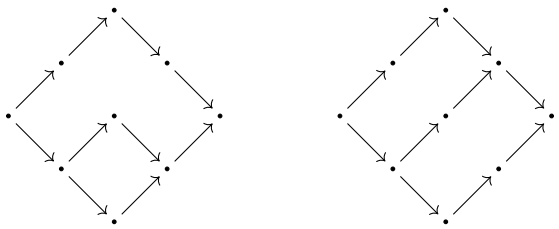
of loops and unfolding is done through the inductive nature of positions. We feel, that is instead possible of defining cubes of programs inductively from cubes of their subprograms. Results from Lemma 3.3.15, Lemma 3.3.16 and Lemma 4.4.22 even hint at the fact that a certain notion of maximality could be preserved by these constructions.

For programs with nested loops, there is still much work to be done. While most of the proofs are independent (or could easily be extended from the case) of nesting loops, some properties (Definition 4.2.13, Lemma 4.3.16) would need some work to get right. The notion of trivial loop is already defined with this extension in mind and should prove useful in an eventual extension.

Abstracting properties of syntactic semantics

As announced before, most of the work in Chapter 4 is still in its early stages and the proofs could benefit from some refining. For now, we quite brutally apply the inductive nature of our programs to the proofs and manage to extract the results, with very few results exploiting known facts in graph theory. However, we do believe that there is a nice general setting in which the graphs associated to state space of programs exists.

Indeed, as already hinted at by Lemma 4.4.16 and the proof of Lemma 4.2.14, there is a certain notion of "well-bracketedness" to the graphs of sequential programs: given a main path π, any pair of paths that split from a vertex must wait for any pair of paths that split further along π to rejoin before they can rejoin themselves. Furthermore, paths are confluent (i.e. they always rejoin). On the left below we have an example of a "well-bracketed" graph and on the right, one that does not have such a property.



This is due to the fact that we do not have `jump` instructions in our language, so that when going through a conditional branching we cannot reach the other branches. One can notice that this also extends to loops, and is what gives us the nice characterization of maximal covers (Proposition 4.3.21) and the cutting properties than ensue from it.

Conservative regions and Alexandroff topology

In our works we had a lot of trouble in how to deal with the core of the loop. We had to force intervals to somehow spread inside the loop, as we feel it is very important to differentiate the different states near the core of the loop (being able to enter, being inside, being able to leave, having left…). We dealt with this problem by forcibly separating these positions through the consideration of conservative regions. This forced the existence of some states ($\bot$ for $P;Q$ that should not differ from $\bot;\bot$ in principle but whose distinction is important in our proofs when considering loops).

Alexandroff spaces [3] are topological space such that every point has a minimal neighbourhood. For graphs, if we define the subset of its set of vertices E to be the opens,

we define an Alexandroff space, where the minimal neighbourhoods are the edges and stars (a vertex with all initial and terminal edges for that vertex). The fact that minimal neighbourhoods are stars and that we mostly consider edges might solve our problem of discretization around the core of the loops.

Instead of considering positions as we did, one could also look at this topology, which can be defined inductively on programs and could join some nice tools from directed topological spaces and the implementability of our methods.

Equivalence between topological and syntactic models

Most tools developed in the case of syntactic models were heavily inspired from tools of the directed topological models of concurrent programs. It is then natural to ask if there exists an equivalence between the resulting objects it generates. One good candidate is the category of components associated to a program.

This topological invariant can be constructed from the data of the maximal cover of the topological space, a cubical partition compatible with this cover and the relation $\lhd$ (Definition 1.4.31) (see [18, Section 6], [28] for more details). All of which have an equivalent in our syntactic models. It is very reasonable to then ask ourselves if the notion already defined have the same properties and can be used to define a "category of components". Furthermore, we would also need to know if this "category of components" correspond to the classical notion.

We believe that the links we have stipulated above with the Alexandroff topology on graphs might play an important part, in the equivalences between these models, as a point where these different approach converge.

Extending lifting property to category of components

In Chapter 4, we have seen that maximal cubes of the base program can be seen as a quotient of the 2-unfolding. The same was also proven for the generic cubical partition (the use of the operator Θ when considering the generic partitions might complicate our affairs) and the relation $\lhd$.

If we can define a notion of category of components for syntactic semantics, it would be interesting to see how this property translates. Even though we don't expect it to work with our current definitions, we have high hopes that many insights can be gained from the attempt.

As previously stated it is still unknown what is the correct definition of the category of components of a program with loops, so there is still much to learn in this direction. Thus considering the quotient, or localization of the category of the 2-unfolding might prove itself insightful in finding the correct way to go about this particular problem.

Chapter A
Technical Background

A.1 Category theory

Here we give an informal primer and reminder on the categorical notion used throughout this thesis. For a more in depth explanation on category theory we refer the reader to the wealth of resources available [38], [21]. As we always deal with small categories in the verification of programs, we have limited our presentation to this case.

Definition A.1.1 (Category)**.** A (small) *category* $\mathcal{C}$ consists of the following data:

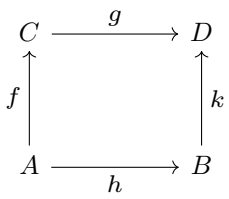
- A set $\mathsf{Obj}(\mathcal{C})$ of objects
- For each pair $x, y \in \mathsf{Obj}(\mathcal{C})$ a collection $\mathsf{Hom}_{\mathcal{C}}(x, y)$, alternatively written $\mathcal{C}(x, y)$, of morphisms (or arrows) from x to y, called a hom-set.
- For each object x a morphism $\mathrm{id}_x \in \mathsf{Hom}_{\mathcal{C}}(x, x)$ called the identity morphism on x
- For each pair $f, g \in \mathcal{C}(y, z) \times \mathcal{C}(x, y)$, a morphism $f \circ g \in \mathcal{C}(x, z)$ called the composite (or composition) of f and g such that:
  - Composition is associative i.e. for any triplet $f, g, h \in \mathcal{C}(y, z) \times \mathcal{C}(x, y) \times \mathcal{C}(w, x)$, we have: $f \circ (g \circ h) = (f \circ g) \circ h$
  - For any $f \in \mathsf{Hom}_{\mathcal{C}}(x, y)$, $f \circ \mathrm{id}_x = f = \mathrm{id}_y \circ f$

We sometimes write $f \colon x \to y$ for $f \in \mathsf{Hom}_{\mathcal{C}}(x, y)$.

Definition A.1.2. Given a morphism $f \colon x \to y$, in a category $\mathcal{C}$. We call x (resp. y) the *source* (resp. *target*) of f

Let g be a second morphism in $\mathcal{C}$, we say that g and f are *co-initial* (resp. *co-final*) if they have the same source (resp. target).

Definition A.1.3. Given the four morphisms $f \colon A \to C$, $g \colon C \to D$, $h \colon A \to B$, $k \colon B \to D$, we say that the following diagram *commutes* when every path (representing composition of morphism) on the diagram is equal, i.e. $g \circ f = k \circ h$



Definition A.1.4. (Opposite Category) Given a category $\mathcal{C}$, the *opposite category* $\mathcal{C}^{\mathrm{op}}$ has the same objects as $\mathcal{C}$, but a morphism $f\colon x \to y$ in $\mathcal{C}^{\mathrm{op}}$ is the same as a morphism $f\colon y \to x$ in $\mathcal{C}$, and a composite of morphism $g \circ f$ in $\mathcal{C}^{\mathrm{op}}$ is defined to be the composite $f \circ g \in \mathcal{C}$

Definition A.1.5 (Isomorphism)**.** Given a category $\mathcal{C}$, and $f \in \mathcal{C}(x, y)$. f is called an *isomorphism* if there exists a morphism f^{-1} such that

$$f \circ f^{-1} = \mathrm{id}_y \qquad\qquad f^{-1} \circ f = \mathrm{id}_x$$

In this case, $f^{-}1$ is unique and called the inverse of f.

Definition A.1.6 (Groupoid)**.** A *groupoid* is a (small) category in which all morphisms are isomorphism

Definition A.1.7 (Functor)**.** A *functor* F from a category $\mathcal{C}$ to a category $\mathcal{D}$ is a map sending each object $x \in \mathcal{C}$ to an object $F(x) \in D$ and each morphism $f : x \to y$ in $\mathcal{C}$ to morphism $F(f) : F(x) \to F(y)$ in $\mathcal{D}$, such that:

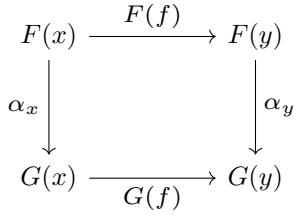
- F preserves composition: $F(g \circ f) = F(g) \circ F(f)$ whenever the left-hand side is well-defined
- F preserves identity morphisms: for each object $x \in C$, $F(1_x) = 1_{F(x)}$.

Alternatively, a functor $f\colon \mathcal{C} \to \mathcal{D}$ sends commutative diagrams in $\mathcal{C}$ to commutative diagrams in $\mathcal{D}$

Definition A.1.8. The category **Cat** is the category which has:

- As objects, all (small) categories
- As arrows of $\mathsf{Hom}_{\mathbf{Cat}}(\mathcal{C}, \mathcal{D})$ all functors $F\colon \mathcal{C} \to \mathcal{D}$
- As composition the evident composition of functors

Definition A.1.9 (Natural transformation)**.** Given two categories $\mathcal{C}, \mathcal{D}$ and two functors $F, G\colon \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\alpha\colon F \Rightarrow G$ is a family of maps $(\eta_x\colon F(x) \to G(x))_{x \in \mathsf{Obj}(\mathcal{C})}$ such that for any morphism $f\colon x \to y$, the following diagram commutes:



If each $eta_x \colon F(x) \to G(x)$ is an isomorphism in D, η is a natural isomorphism, and we write $F \backsimeq G$.

Definition A.1.10 (Presheaves)**.** Given a small category $\mathcal{S}$, a *presheaf* on $\mathcal{C}$ is a functor

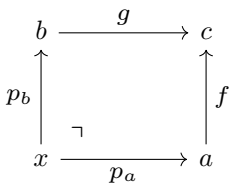
$$F \colon \mathcal{C}^{op} \to \mathbf{Set}$$

The *category of presheaves* on $\mathcal{C}$, denoted $[\mathcal{C}^{op}, \mathbf{Set}]$, $\mathbf{Set}^{\mathcal{C}^{op}}$, or simply $\widehat{\mathcal{C}}$ has:

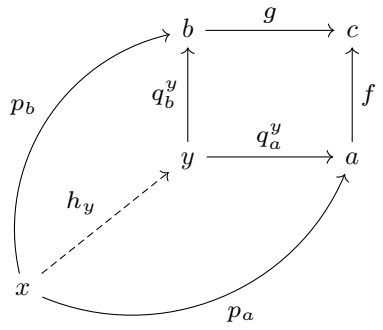
- functors $F \colon \mathcal{C}^{op} \to \mathbf{Set}$ as objects,
- natural transformations between such functors as morphisms

Definition A.1.11 (Pullback, pushout)**.** Given a category $\mathcal{C}$, with $f \colon a \to c$ and $g \colon b \to c$, two co-terminal arrows of $\mathcal{C}$. A *pullback* of f and g consists of an object x, together with arrows $p_a \colon x \to a$ and $p_b \colon x \to b$ such that:

- The following diagram commutes:



- For any object y together with a pair of arrows q_a^y, q_b^y verifying the first condition, there exists a unique $h \colon x \to y$ making the following diagram commute:



A *pushout* in $\mathcal{C}$ is a pullback in $\mathcal{C}^{op}$

Definition A.1.12 (Product Category)**.** Given a category $\mathcal{C}$ and any set of its objects $\{X_i\}_{\{i \in I\}}$, the *product* of $\{X_i\}_{\{i \in I\}}$ is, if it exists an object denoted

$$\prod_{i \in I} X_i \in \mathcal{C}$$

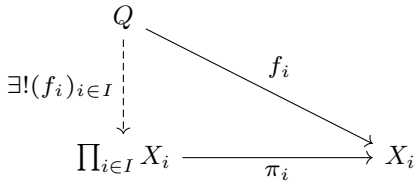
and equipped with morphisms

$$\pi_j \colon \prod_{i \in I} X_i \to X_j$$

called *projections* for all $j \in I$, such that for any family of morphisms $\{f_i \colon Q \to X_i\}_{i \in I}$, there exists a unique morphism

$$(f_i)_{i \in I} \colon Q \to \prod_{i \in I} X_I$$

such that all the following diagrams commutes:



Example A.1.13. In **Cat**, all products are defined (we say that **Cat** has all products). The product of two categories $\mathcal{C}, \mathcal{D}$ is the category $\mathcal{C} \times \mathcal{D}$ which has:

- as set of objects the cartesian product $\mathsf{Obj}(\mathcal{C}) \times \mathsf{Obj}(\mathcal{D})$
- as hom-sets $\mathsf{Hom}_{\mathcal{C} \times \mathcal{D}}(X \times U, Y \times V) = \mathsf{Hom}_{\mathcal{C}}(X, Y) \times \mathsf{Hom}_{\mathcal{D}}(U, V)$
- as composition, the natural composition on pairs i.e. $(g, k) \circ (f, h) = (g \circ f, k \circ h)$

Definition A.1.14 (Product Preserving Functor)**.** Given a functor $F \colon \mathcal{C} \to \mathcal{D}$ and $(X_i)_{i \in I}$ a set of objects in $\mathcal{C}$ which admits a product $\prod_{i \in I} X_i$, equipped with projections $\pi_i \colon \prod_{i \in I} X_i \to X_i$. We say that F *preserves* this product if $F(\prod_{i \in I} X_i)$, with projections $F(\pi_i)$ is the product of the objects $(F(X_i)_{i \in I})$

Definition A.1.15 (Hom-Functor)**.** Given a small category, its *hom-functor* is the functor $\mathsf{Hom}_{\mathcal{C}}(-,-) \colon \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$ from the product category of $\mathcal{C}^{op}$ and $\mathcal{C}$ to the category of sets, which sends:

- each object $(x, y) \in \mathcal{C}^{op} \times \mathcal{C}$ to the hom-set $\mathsf{Hom}_{\mathcal{C}}(x, y)$
- each morphism $(x, y) \to (w, z)$ of $\mathcal{C}^{op} \times \mathcal{C}$, i.e. each pair of morphism $f^o p \colon x \to w$, $g \colon y \to z$ to the function

$$\begin{aligned} &\mathsf{Hom}_{\mathcal{C}}(x, y) \to \mathsf{Hom}_{\mathcal{C}}(w, z) \\ &\{q \colon x \to y\} \mapsto \{g \circ q \circ f \colon w \to z\} \end{aligned}$$

Definition A.1.16 (Adjoint Functors)**.** Given two categories $\mathcal{C}, \mathcal{D}$ and two functors $L \colon \mathcal{C} \to \mathcal{D}$ and $R \colon \mathcal{D} \to \mathcal{C}$ We say that L is *left adjoint* to R (or R right adjoint to L) when there exists a natural isomorphism between the hom-functors of the following form:

$$\mathsf{Hom}_{\mathcal{D}}(L(-), -) \backsimeq \mathsf{Hom}_{\mathcal{C}}(-, R(-))$$

Meaning that for all objects $c \in \mathcal{C}$ and all objects $d \in \mathcal{D}$, there is a bijection of hom-sets:

$$\mathsf{Hom}_{\mathcal{D}}(L(c), d) \xrightarrow{\backsimeq} \mathsf{Hom}_{\mathcal{C}}(c, R(d))$$

which is natural in c and d.

Definition A.1.17 (Full, Faithful, Essentially Surjective)**.** A functor $F\colon \mathcal{C} \to \mathcal{D}$ is called:

- *full* if for each pair of objects $x, y \in \mathcal{C}$, the function $F\colon \mathcal{C}(x,y) \to \mathcal{D}(F(x), F(y))$ between hom-sets is surjective
- *faithful* if for each pair of objects $x, y \in \mathcal{C}$, the function $F\colon \mathcal{C}(x,y) \to \mathcal{D}(F(x), F(y))$ between hom-sets is surjective
- *fully faithful* if it is both full and faithful
- *essentially surjective* if for every object y of $\mathcal{D}$, there is an object x of $\mathcal{C}$ such that $F(x)$ is isomorphic to y.

Definition A.1.18 (Subcategory)**.** Given a category $\mathcal{C}$, a *subcategory* $\mathcal{D}$ of $\mathcal{C}$ is a subset of set the objects of $\mathcal{C}$ and a subset of the set of morphisms of $\mathcal{C}$ such that

- If $f\colon x \to y$ is in $\mathcal{D}$ so are x and y
- If $f\colon x \to y$ and $g\colon y \to z$ are in $\mathcal{D}$, so is their composite $g \circ f$.
- If x is an object of $\mathcal{D}$, then id_x is in $\mathcal{D}$

Then $\mathcal{D}$ is a category in its own right and the inclusion $\mathcal{D} \hookrightarrow \mathcal{C}$ is a functor, called the *inclusion functor*.

Definition A.1.19 (Skeleton)**.** The *skeleton* of a category $\mathcal{C}$ is the unique subcategory $\mathcal{D}$ of $\mathcal{C}$ such that:

- The inclusion functor $\iota\colon \mathcal{D} \hookrightarrow \mathcal{C}$ is full,
- The inclusion functor ι is essentially surjective,
- $\mathcal{D}$ is skeletal, i.e. no two distinct objects of $\mathcal{D}$ are isomorphic.

Definition A.1.20 (Discrete Fibration)**.** A functor $F\colon \mathcal{C} \to \mathcal{B}$ is a *discrete fibration* if for every object c in $\mathcal{C}$, and every morphism of the form $g\colon b \to F(c)$ in B there is a unique morphism $h\colon d \to c$ such that $F(h) = g$.

A.2 Order theory

Definition A.2.1 (Pre-order)**.** A *pre-order* $(X, \preceq)$ is a set X equipped with a relation $\preceq$ defined on $X \times X$ such that:

- $\preceq$ is transitive i.e. for any $x, y, z \in X$, $x \preceq y$ and $y \preceq z$ implies $x \preceq z$
- $\preceq$ is reflexive i.e. $x \preceq x$ for all $x \in X$

Definition A.2.2 (Partial order)**.** A *partial order* $(X, \leq)$ is a set X equipped with a relation $\leqq$ defined on $X \times X$ such that:

- $(X, \leq)$ is a pre-order
- $\leq$ is antisymmetric i.e. for any $x, y \in X$, $x \leq y$ and $y \leq x$ implies $x = y$.

We say that X is a *partially ordered set*, or *poset*.

Definition A.2.3 (Bounded Preorder)**.** A *bounded pre-order* $(X, \bot, \top)$ is a pre-order X equipped with two distinguished elements $\bot, \top$ such that

- For any $x \in X$, $\bot \preceq x$ and $x \not\preceq \bot$
- For any $x \in L$, $x \preceq \top$ and $\top \not\preceq x$

$\bot$, $\top$ can be seen respectively as the smallest and greatest element of the set X.

Definition A.2.4 (Lattice)**.** A *lattice* $(L, \leq, \wedge, \vee)$ is a poset $(L, \leq)$ such that each two-element subset $\{a, b\} \subseteq L$ has:

- a least upper bound (or *join*) in L, denoted $a \vee b$,
- and a greatest lower bound (or *meet*) in L, denoted $a \wedge b$

This is equivalent to requiring that every finite subset $X \subseteq L$ has a join (resp. meet) denoted $\bigvee X$ (resp. $\bigwedge X$)

Definition A.2.5 (Bounded lattice)**.** A *bounded* lattice $(L, \bot, \top)$ is a lattice equipped with two distinguished elements $\bot, \top$ such that

- For any $x \in L$, $x \wedge \bot = \bot$
- For any $x \in L$, $x \vee \top = \top$

Equivalently, $(L, \bot, \top)$ is a bounded lattice if and only if $(L, \bot, \top)$ is a bounded pre-order, where $\bot$ and $\top$ are respectively the smallest and greatest elements.

Definition A.2.6 (Complete lattice)**.** A *complete* lattice L is a lattice where *any* subset $X \subseteq L$ has a meet $\bigvee X$ and join $\bigwedge X$

By definition, a complete lattice L is bounded, with $\bot = \bigwedge L$, $\top = \bigvee L$

Definition A.2.7 (Monotone functions)**.** Given two posets $(\mathcal{C}, \leq_{\mathcal{C}})$, $(\mathcal{D}, leq_{\mathcal{D}})$ a function $f \colon \mathcal{C} \to \mathcal{D}$ is *monotone* when for every $x, y \in \mathcal{C}$, $x \leq_{\mathcal{C}} y$ implies $f(x) \leq_{\mathcal{D}} f(y)$

Definition A.2.8. Every preorder X can be seen as a category, with $\mathsf{Obj}(X) = X$ and each $\mathsf{Hom}_X(x, y)$ defined to contain a single element if $x \leq y$, and otherwise it is empty. We write **Pos** for the category of partially ordered sets.

With this definition, a monotone function $f \colon \mathcal{C} \to \mathcal{D}$ is a functor from $\mathcal{C}$ to $\mathcal{D}$, i.e. a morphism in **Pos**

Definition A.2.9 (Adjunction in posets)**.** Given two posets $\mathcal{C}$, $\mathcal{D}$ and a pair of monotone functions $l \colon CatC \to \mathcal{D}$, $r \colon \mathcal{D} \to CatC$, then l is left adjoint to r if and only if for any $c, d \in \mathcal{C} \times \mathcal{D}$

$$l(x) \leq y \iff x \leq r(y)$$

Alternatively, we say that there is a monotone Galois connection between $\mathcal{C}$ and $\mathcal{D}$.

Theorem A.2.10 (Adjunct functor theorem)**.** *Given two posets $\mathcal{C}$, $\mathcal{D}$ and a pair of monotone functions $l\colon CatC \to \mathcal{D}$, $r\colon \mathcal{D} \to CatC$ such that l is left-adjoint to r, then*

$$l(c) = \bigvee\{d \in \mathcal{D} \mid r(d) = c\} \qquad r(d) = \bigwedge\{c \in \mathcal{C} \mid l(c) = d\}$$

Definition A.2.11 (Boolean algebra)**.** A *Boolean algebra* $(B, \wedge, \vee)$ is a bounded lattice $(B, \wedge, \vee)$ where $\wedge$ and $\vee$ verify the following additional properties:

- $\wedge$ and $\vee$ distribute over each other i.e. for any elements $x, y, z \in B$:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \qquad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$$

- any element $x \in B$ has a complement i.e. there exists $x^{\mathrm{c}} \in B$ such that:

$$x \wedge x^{\mathrm{c}} = \bot \qquad x \vee x^{\mathrm{c}} = \top$$

Definition A.2.12 (Atomic boolean algebra)**.** Given a Boolean algebra B, an *atom* is an element $a \in B$, such that for any $b \in B$, $b \leq a$ implies $b = \bot$ or $b = a$.

A Boolean algebra is atomic if for any element $b \in B$, there exists a set $\{a_i\}_{i \in I_b}$ of atoms of B such that $b = \bigvee\{a_i\}_{i \in I_b}$.

A.3 Topology

Definition A.3.1 (Topological space)**.** Given a set X, a topology τ on X is a collection $\tau \subseteq \mathfrak{P}(X)$ of subset of X called the *open subsets* of X, such that:

- τ is closed under finite intersections,
- τ is closed under arbitrary union.

A topological space (X, τ) is a set X equipped with a topology τ

Definition A.3.2 (Metric space)**.** A metric space (X, d_X) is a set X equipped with a *distance function* $d\colon x \times X \to [0, +\infty[$ such that for all $x, y, z \in X$:

- $d(x, y) = 0$ if and only if $x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

To any metric space X, we can assign a topology where the open sets of X are the open balls $B_r(x) = \{y \in X \mid d(x, y) < r\}$ for all $x \in X$ and all $r \in \mathbb{R}^+$

Definition A.3.3 (Euclidian topology)**.** The euclidean topology on $\mathbb{R}^n$, is the topology associated to the metric space $(\mathbb{R}^n, d)$, where $d(x, y) = \|x - y\|$, with $\|\cdot\|$ the standard euclidean norm:

$$\|(x_1, \ldots, x_n)\| = \sqrt{x_1^2 + \cdots + x_n^2}$$

Definition A.3.4 (Continuous map)**.** A *continuous* function $f\colon (X, \tau_X) \to (Y, \tau_Y)$ between topological spaces is a function $f\colon X \to Y$ between the underlying sets, such that for every open V, the inverse image by f is an open of X i.e. for all $V \subseteq Y$

$$V \in \tau_Y \text{ implies } f^{-1}(V) \in \tau_X$$

Definition A.3.5 (Paths)**.** We write I for the unit interval $[0, 1] \subseteq \mathbb{R}$ equipped with the euclidean topology. A *path* on X is a continuous map $f\colon I \to X$.

Definition A.3.6 (Concatenation of maps)**.** Given f, g two paths on a topological space X, such that $f(1) = g(0)$ we call concatenation of g and f, the path $g \cdot f$ on X, defined as follows:

$$f \cdot g(t) = \begin{cases} f(2t) & \text{if } 0 \leq t \leq 1/2 \\ g(2t-1) & \text{otherwise} \end{cases}$$

Definition A.3.7 (Homotopy)**.** Given two continuous maps between topological spaces $f, g\colon X \to Y$. A homotopy from f to g is a continuous map $h\colon I \times X \to Y$ such that $h(0, -) = f$ and $h(1, -) = g$. When such an h exists, the maps f and g are said to be homotopic, written $f \sim g$.

The homotopy relation $\sim$ defines an equivalence relation

Definition A.3.8 (Fundamental groupoid)**.** The fundamental groupoid $\Pi_1(X)$ associated to the topological space X is the groupoid:

- whose objects are the points of X,
- whose morphisms are the equivalence classes of endpoint preserving homotopy on paths of X,
- whose composition is defined by concatenation of paths.

Bibliography

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2:117–126, 1987.

[3] Francisco G Arenas. Alexandroff spaces. *Acta Math. Univ. Comenianae*, 68(1):17–25, 1999.

[4] Martin Arkowitz. *Introduction to homotopy theory*. Springer Science & Business Media, 2011.

[5] Thibaut Balabonski and Emmanuel Haucourt. A geometric approach to the problem of unique decomposition of processes. *CoRR*, abs/1004.2780, 2010. `arXiv:1004.2780`.

[6] Marek A. Bednarczyk, Andrzej M. Borzyszkowski, and Wieslaw Pawlowski. Generalized congruences-epimorphisms in cat. *Theory and Applications of Categories*, 5(11):266–280, 1999.

[7] Edmund M Clarke. Model checking–my 27-year quest to overcome the state explosion problem. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 182–182. Springer, 2008.

[8] Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[9] Cosynus. Alcool, 2018. Available at `http://www.lix.polytechnique.fr/cosynus/alcool/`.

[10] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295. Springer, 1992.

[11] Edsger W Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[12] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta informatica*, 1:115–138, 1971.

[13] Edsger W. Dijkstra. *Cooperating Sequential Processes*, pages 65–138. Springer New York, New York, NY, 2002. doi:10.1007/978-1-4757-3472-0_2.

[14] E.W. Dijkstra. Co-operating sequential processes. In *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys*, pages 43–112, United States, 1968. Academic Press Inc.

[15] Allen B Downey. *The little book of semaphores*. 2005.

[16] Jannik Dreier, Cristian Ene, Pascal Lafourcade, and Yassine Lakhnech. On Unique Decomposition of Processes in the Applied π-Calculus. In *16th International Conference on Foundations of Software Science and Computational Structures (FOSSACS 2013), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2013)*, Rome, Italy, March 2013. URL: https://inria.hal.science/hal-01338002, doi:10.1007/978-3-642-37075-5_4.

[17] Lisbeth Fajstrup. Loops, ditopology and deadlocks. *Mathematical Structures in Computer Science*, 10(4):459–480, 2000. doi:10.1017/S0960129500003157.

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

[19] Lisbeth Fajstrup, Eric Goubault, and Martin Raußen. Detecting deadlocks in concurrent systems. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR'98 Concurrency Theory*, pages 332–347, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[20] Lisbeth Fajstrup, Martin Raussen, Éric Goubault, and Emmanuel Haucourt. Components of the fundamental category. *Applied Categorical Structures*, 12:81–108, 2004.

[21] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality*. Cambridge University Press, 2019.

[22] Peter Gabriel and Michel Zisman. *Calculus of fractions and homotopy theory*, volume 35. Springer Science & Business Media, 2012.

[23] Alfred Geroldinger and Franz Halter-Koch. *Non-unique factorizations: Algebraic, combinatorial and analytic theory*. CRC Press, 2006.

[24] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[25] Marco Grandis. *Directed algebraic topology: models of non-reversible worlds*, volume 13. Cambridge University Press, 2009.

[26] Junji Hashimoto. On direct product decomposition of partially ordered sets. *Annals of Mathematics*, pages 315–318, 1951.

[27] Allen Hatcher. *Algebraic topology*. Cambridge University Press, 2005.

[28] Emmanuel Haucourt. Categories of components and loop-free categories. *Theory and Applications of Categories*, 16(27):736–770, 2006.

[29] Emmanuel Haucourt. *Some Invariants of Directed Topology towards a Theoretical Base for a Static Analyzer Dealing with Fine-Grain Concurrency*. PhD thesis, Université Paris 7-Denis Diderot, 2016.

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

[31] Emmanuel Haucourt and Nicolas Ninin. Unique decomposition of homogeneous languages and application to isothetic regions. *Mathematical Structures in Computer Science*, 29(5):681–730, 2019. doi:10.1017/S0960129518000294.

[32] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[33] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 254–266. IEEE, 1977.

[34] Joseph B Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American mathematical society*, 95(2):210–225, 1960.

[35] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9):690–691, 1979.

[36] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[37] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, Jérôme Vouillon, et al. OCaml system release 4.02: Documentation and user's manual (2014), 2017.

[38] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[39] Saunders MacLane and Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, 2012.

[40] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall Englewood Cliffs, 1989.

[41] Robin Milner and Faron Moller. Unique decomposition of processes. *Theoretical Computer Science*, 107(2):357–363, 1993. URL: https://www.sciencedirect.com/science/article/pii/030439759390176T, doi:10.1016/0304-3975(93)90176-T.

[42] Samuel Mimram and Aly-Bora Ulusoy. Sparkling, 2021. Available at https://smimram.github.io/sparkling/.

[43] Samuel Mimram and Aly-Bora Ulusoy. Syntactic regions for concurrent programs. In *MFPS*, 2021.

[44] Antoine Miné. Static analysis of run-time errors in embedded critical parallel c programs. In *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20*, pages 398–418. Springer, 2011.

[45] Tadasi Nakayama and Junji Hashimoto. On a problem of G. Birkhoff. In *On a problem of G. Birkhoff*, 1950.

[46] Nicolas Ninin. *Factorisation des régions cubiques et application à la concurrence*. PhD thesis, École polytechnique, 12 2017.

[47] Ganesan Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming languages and Systems (TOPLAS)*, 22(2):416–430, 2000.

[48] Sylvain Schmitz and Philippe Schnoebelen. Algorithmic aspects of WQO theory. 2012.

[49] Bernd S.W. Schröder. Ordered sets. *Springer*, 29:30, 2003.

[50] Jaroslav Ŝevčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 43–54, 2011.

[51] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 209–220, 2015.

[52] Antti Valmari. A stubborn attack on state explosion: Computer-aided verification 90. *EM Clarke and RP Kurshan (Eds.)*, (3):25–41, 1990.

[53] Rob J. van Glabbeek. On the expressiveness of higher dimensional automata. *Theoretical computer science*, 356(3):265–290, 2006.

[54] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.